National Research Council Canada
Institute for Information Technology

Conseil national de recherches Canada
Institut de Technologie de l'information

ERB-1080

NRC·CNRC

Validating Object-oriented Design Metrics on a Commercial Java Application

Daniela Glasberg, Khaled El Emam,
Walcelio Melo, and Nazim Madhavji
September 2000

Canada

NRC 44146

National Research Council Canada
Institute for Information Technology

Conseil national de recherches Canada
Institut de Technologie de l'information

Validating Object-oriented Design Metrics on a Commercial Java Application

Daniela Glasberg, Khaled El Emam,
Walcelio Melo, and Nazim Madhavji
September 2000

Copyright 2000 by
National Research Council of Canada

Permission is granted to quote short excerpts and to reproduce figures and tables from this report, provided that the source of such material is fully acknowledged.

Validating Object-Oriented Design Metrics on a Commercial Java Application

Daniela Glasberg[1]
Khaled El Emam[2]
Walcelio Melo[3]
Nazim Madhavji[4]

Abstract

Many of the object-oriented metrics that have been developed by the research community are believed to measure some aspect of complexity. As such, they can serve as leading indicators of problematic classes, for example, those classes that are most fault-prone. If faulty classes can be detected early in the development project's life cycle, mitigating actions can be taken, such as focused inspections. Prediction models using design metrics can be used to identify faulty classes early on. In this paper, we present a cognitive theory of object-oriented metrics and an empirical study which has as objectives to formally test this theory while validating the metrics and to build a post-release fault–proneness prediction model. The cognitive mechanisms which we apply in this study to object-oriented metrics are based on contemporary models of human memory. They are: familiarity, interference, and fan effects. Our empirical study was performed with data from a commercial Java application. We found that Depth of Inheritance Tree (DIT) is a good measure of familiarity and, as predicted, has a quadratic relationship with fault–proneness. Our hypotheses were confirmed for Import Coupling to other classes, Export Coupling and Number of Children metrics. The Ancestor based Import Coupling metrics were not associated with fault-proneness after controlling for the confounding effect of DIT. The prediction model constructed had a good accuracy. Finally, we formulated a cost savings model and applied it to our predictive model. This demonstrated a 42% reduction in post-release costs if the prediction model is used to identify the classes that should be inspected.

1 Introduction

A considerable number of object-oriented metrics have been developed by the research community. The basic premise behind these metrics is that they capture some elements of object-oriented software complexity. Examples of metrics include those defined in (Abreu and Carapuca, 1994; Benlarbi and Melo, 1999; Briand et al., 1997; Cartwright and Shepperd, 2000; Chidamber and Kemerer, 1994; Henderson-Sellers, 1996; Li and Henry, 1993; Lorenz and Kidd, 1994; Tang et al., 1999). While many of these metrics are based on seemingly good ideas about what is important to measure in object-oriented software to capture its complexity, it is still necessary to empirically validate them.

---

[1] School of Computer Science McGill University, 3480 University Street, McConnell Engineering Building, Montreal, Quebec, Canada H3A 2A7 dglasb@cs.mcgill.ca

[2] National Research Council of Canada, Institute for Information Technology, Building M-50, Montreal Road, Ottawa, Ontario, Canada K1A OR6, Khaled El-Emam@nrc.ca

[3] Oracle Brazil, SCN Qd. 2 Bl. A, Ed. Corporate, S. 604, 70712-900 Brasilia, DF, Brazil wmelo@br.oracle.com

[4] School of Computer Science McGill University, 3480 University Street, McConnell Engineering Building, Montreal, Quebec, Canada H3A 2A7 madhavji@cs.mcgill.ca

Empirical validation involves demonstrating an association between the metric under study and other measures of important external attributes (ISO/IEC-14598-1, 1996). An external attribute is concerned with how the product relates to its environment (Fenton, 1991). Examples of external attributes are testability, reliability and maintainability. Practitioners, whether they are developers, managers, or quality assurance personnel, are really concerned with the external attributes. However, they cannot measure many of the external attributes directly until quite late in a project's or even a product's life cycle. Therefore, they can use product metrics as leading indicators of the external attributes that are important to them. By having good leading indicators, it is possible to predict the external attributes and take early action if the predictions do not fit a project's objectives. For instance, if we know that a certain coupling metric is a good leading indicator of maintainability as measured in terms of the effort to make a corrective change, then we can minimize coupling during design because we know that in doing so we are also increasing maintainability.

In this paper we focus on *empirically validating* a set of object-oriented design metrics developed by Chidamber and Kemerer (Chidamber and Kemerer, 1994) and Briand et al. (Briand et al., 1997). The study was performed with data collected from a commercial Java application that implements an XML editor. The external attribute that we measure for our study is reliability. Reliability can be measured in different ways. We measure it in terms of the incidence of a post-release fault in a class. This is termed the *fault-proneness* of a class.

While previous empirical validation studies of these metrics have been performed (Briand et al., 1997; Briand et al., 1998a; Briand et al., 2000; El-Emam et al., 1999, 2000b; El-Emam et al., 2001b), our current work is intended to replicate these studies in order to provide accumulating evidence of the metrics' validity. Furthermore, we make a number of additional contributions to the above studies:

- We present a detailed cognitive theory to justify the above metrics. Based on this theory we state a number of precise hypotheses relating the design metrics to fault-proneness. Evidence supporting the hypotheses suggests possible reasons *why* certain metrics are leading indicators of fault-proneness. To our knowledge, this is the first attempt at postulating a detailed cognitive theory for object-oriented metrics.
- An object-oriented metrics cost-benefit model is formulated. This model can be used to evaluate the cost savings from using the design metrics to predict which classes are most fault-prone, and target these classes for inspection. The model is then applied to illustrate its utility.

Our results indicate that many of the design metrics are indeed associated with fault-proneness. The directions of the associations conform to what we predicted from the cognitive theory. This is encouraging as now we have evidence for some cognitive mechanisms to explain the impact of object-oriented metrics. We also built a fault-proneness prediction model using a subset of the validated metrics. The fault-proneness prediction model was found to have high accuracy, and the post-release cost savings from the application of the prediction model were estimated to be 42%. This means that had the prediction model been used to identify the classes to inspect (i.e., those with the highest predicted fault-proneness), then 42% of the post-release costs (i.e., costs associated with dealing with faults discovered by customers) would have been saved. For many organizations such savings are nontrivial, and can free up resources to add features to their products.

In the next section we present a theory that provides mechanisms explaining why the object-oriented metrics we study would be associated with fault-proneness, and give an overview of the metrics we study. Section 3 presents our research method in detail, and our results are presented and discussed in Section 4. We conclude the paper in Section 5 with a summary and directions for future work.

2 Background

Below we present a detailed cognitive theory of object-oriented metrics. The theory explains the mechanisms as to why object-oriented metrics are associated with fault-proneness. By giving the mechanisms, it predicts which metrics are likely to be associated with class fault-proneness, and the direction and functional form of the association. In our study we then empirically test these predictions.

It is important to articulate this theory in detail so that it can be tested. If the theory can be supported empirically, then it is possible to devise better metrics (i.e., with greater predictive power or that can be collected early in the life cycle). Furthermore, if we understood the mechanism then it is plausible that we can improve the design and programming process to avoid the introduction of faults altogether. It is also possible that hypotheses derived from a cognitive theory *cannot* be empirically supported. In such a case cognitive complexity may not be the reason why certain structural properties of object-oriented applications are problematic, and we have to focus our attention on finding the mechanism of fault introduction.

2.1 Overview of the Cognitive Theory

In order for object-oriented metrics to be associated with fault-proneness, there has to be a mechanism causing this effect. In the software engineering literature, this mechanism is believed to be *cognitive complexity*. For instance, Munson and Khoshgoftaar (Munson and Khoshgoftaar, 1992) state that "There is a clear intuitive basis for believing that complex programs have more faults in them than simple programs". It has been argued that a detailed cognitive model is a necessary basis for developing software product metrics (Bergantz and Hassell, 1991; Ehrlich and Soloway, 1984). A further argument has been made that unless there is an understanding of the cognitive demands that software places on developers, then only surface features of the software will be measured, and such surface "complexity" features will not be effective (Sebrechts and Black, 1982). This implies that a cognitive theory of object-oriented metrics that explains how and why certain metrics are associated with fault-proneness is necessary.

Figure 1 depicts the cognitive theory that we describe. The rationale for the theory is the common belief that the structural properties of a software component (such as its coupling and inheritance relationships) have an impact on its cognitive complexity (Briand et al., 2000; Cant et al., 1994; Cant et al., 1995; Henderson-Sellers, 1996).

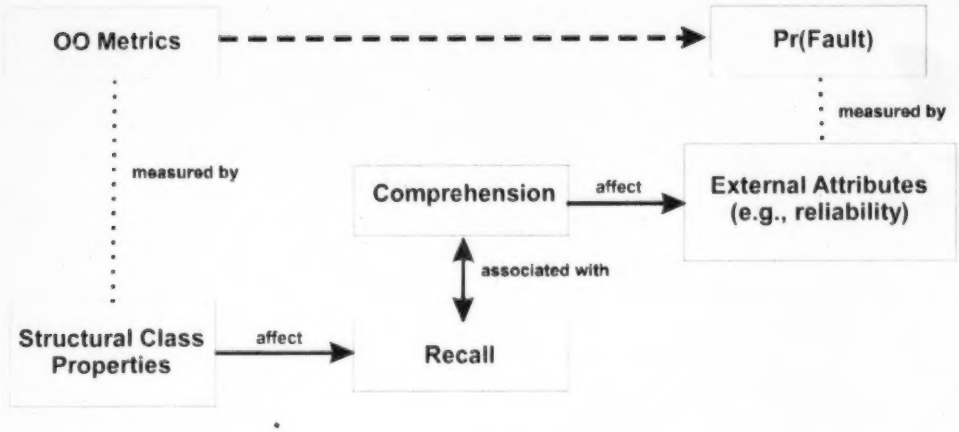


Figure 1: Overview of object-oriented metrics theory. The dotted lines indicate a relationship between specific metrics and the concepts that they are measuring. The dashed line depicts the relationship that is actually tested in validation studies. The filled lines indicate hypothesized relationships that explain why structural class properties are associated with external attributes.

One way to operationalize cognitive complexity is to equate it with the ease of comprehending an object-oriented application. If classes are difficult to comprehend then there is a greater probability that faults will be introduced during development. For example, modifications due to requirements changes or fixing faults found during inspections and testing will be error-prone. Furthermore, it will be more difficult to detect faults during the development fault detection activities, such as inspections. In both cases there is a positive association between comprehension and reliability.

A number of studies indicate that certain features of object-oriented application impede their comprehension. Widenbeck et al. (Wiedenbeck et al., 1999) make a distinction between program functionality at the local level and at the global (application) level. At the local level they argue that the object-oriented paradigm's concept of encapsulation ensures that methods are bundled together with the data on which they operate, making it easier to construct appropriate mental models and specifically to understand a class' individual functionality. At the global level, functionality is dispersed among many interacting classes, making it harder to understand what the program is doing. They support this in an experiment where they found that the number of correct answers for subjects comprehending a C++ program (with inheritance) on questions about its functionality was not much better than guessing. In another experimental study with students and professional programmers (Boehm-Davis et al., 1992), Boehm-Davis et al. compared maintenance time for three pairs of functionally equivalent programs (implementing three different applications amounting to a total of nine programs). Three programs were implemented in a straight serial structure (i.e., one main function, or monolithic program), three were implemented following the principles of functional decomposition, and three were implemented in the object-oriented style, but without inheritance. In general, it took the students

more time to change the object-oriented programs, and the professionals exhibited the same effect, although not as strongly. Furthermore, both the students and professionals noted that they found that it was most difficult to recognize program units in the object-oriented programs, and the students felt that it was also most difficult to find information in the object-oriented programs.

Typically, structural properties that capture dependencies among classes are believed to exert significant influence on comprehension, for example, *coupling* and *inheritance*. Coupling metrics characterize the static[5] usage dependencies among the classes in an object-oriented system (Briand et al., 1999a). Inheritance is also believed to play an important role in the complexity of object-oriented applications. Coupling and inheritance capture dependencies among the classes.

Below we present evidence indicating that there is typically a profusion of dependencies in object-oriented applications, and that these dependencies have an impact on understandability and fault-proneness.

The object-oriented strategies of limiting a class' responsibility and reusing it in multiple contexts results in a profusion of small classes in object-oriented systems (Wilde et al., 1993). For instance, Chidamber and Kemerer (Chidamber and Kemerer, 1994) found in two systems studied[6] that most classes tended to have a small number of methods (0-10), suggesting that most classes are relatively simple in their construction, providing specific abstraction and functionality. Another study of three systems performed at Bellcore[7] found that half or more of the methods are fewer than four Smalltalk lines or two C++ statements, suggesting that the classes consist of small methods (Wilde et al., 1993). Many small classes imply many interactions among the classes and a distribution of functionality across them.

It has been stated that "Inheritance gives rise to distributed class descriptions. That is, the complete description for a class D can only be assembled by examining D as well as each of D's superclasses. Because different classes are described at different places in the source code of a program (often spread across several different files), there is no single place a programmer can turn to get a complete description of a class" (Leijter et al., 1992). While this argument is stated in terms of source code, it is not difficult to generalize it to design documents.

Cant et al. (Cant et al., 1994) performed an empirical study whereby they compared subjective ratings by two expert programmers of the complexity of understanding classes with objective measures of dependencies in an object-oriented system. Their results demonstrate a correlation between the objective measures of dependency and the subjective ratings of understandability. In an experience report on learning and using Smalltalk (Nielsen and Richards, 1989), the authors found that the distributed nature of the code causes problems when attempting to understand a system.

Dependencies also make it more difficult to detect faults. For example, Dunsmore et al. (Dunsmore et al., 2000) note that dependencies in object-oriented programs lead to functionality being distributed across many classes. The authors performed a study with student subjects where the subjects were required to inspect a Java program. The results indicated that the most difficult faults to find were those characterized by a delocalization of information needed to fully understand the fault. A subsequent survey of experienced object-oriented professionals indicated that delocalization was perceived to be a major problem in terms of fault introduction and understandability of object-oriented applications.

We postulate that the extent of dependencies has an impact on comprehension. In Figure 1 we show the constructs of this theory. There are four constructs: (i) structural class properties, (ii) recall, (iii) comprehension, and (iv) external attributes. In software engineering work, we only measure the first and fourth constructs. So we have object-oriented metrics to measure

---

[5] Here static means without the actual execution of the application.

[6] One system was developed in C++, and the other in Smalltalk.

[7] The study consisted of analyzing C++ and Smalltalk systems and interviewing the developers for two of them. For a C++ system, method size was measured as the number of executable statements, and for Smalltalk size was measured by uncommented nonblank lines of code.

dependencies among classes, and we have metrics of the incidence of faults to measure reliability. We expect that these two will be related. The reason that they are related is explained by the relations through recall and comprehension. In the appendix (Section 8) we elaborate on the theory in detail, presenting the evidence supporting its formulation.

Research on procedural and object-oriented program comprehension has identified three types of mental models that developers construct and update during comprehension (von Mayrhauser and Vans, 1995a). Developers will frequently switch their attention among these mental models. However, during comprehension it is necessary to search and extract information from the program being comprehended, and to connect information within each of these mental models. This requires the recall of information already extracted to construct the mental models.

Dependencies in an object-oriented artifact make it difficult for someone to recall information about the artifact. This impedes comprehension and results in incomplete mental models. Some commonly known effects in cognitive psychology to explain this are: *interference effects*, *fan effects*, and *familiarity*.

Interference effects occur when a subject learns intervening material after some initial material. The initial material will be more difficult to recall. Fan effects occur when a concept that has been learned has many other concepts associated with it. This leads to that concept being difficult to recall. Familiarity occurs when a concept in memory is repeatedly recalled, and so it becomes easier to recall again.

Based on studies of engineers comprehending an object-oriented system (Burkhardt et al., 1998), we postulate that the way they trace through the class hierarchy when trying to understand the relationship among classes can be mapped to the cognitive effects above. For example, when an engineer is trying to comprehend a class X and encounters an attribute whose type is another class Y, then s/he will proceed to class Y to comprehend what it is doing. This results in an interference effect while comprehending class X. The more class X has connections to other classes, the more difficult it will be to recall information about X due to the fan effect. Furthermore, if class X is an attribute in many other classes, then it will be consulted often during comprehension and therefore will be familiar.

For general recall task as well as for program recall, it has been found that recall is associated with comprehension. The lack of comprehension leads to the introduction of faults, and a difficulty in finding faults during defect detection activities.

It should be noted that this theory is not intended to define all the mechanisms that are believed or that have been shown in the past to have an impact on fault-proneness. But rather, we wish to only focus on the factors that can plausibly *explain* the relationship between structural class properties and reliability. The limitations of this theory are further elaborated in Section 4.10.

2.2 Structural Class Properties and Their Measurement

Different sets of metrics have been developed in software engineering to capture these object-oriented dependencies. We will focus here on metrics that characterize the inheritance hierarchy and coupling among classes.

The metrics used in our study are a subset of the two sets of metrics defined by Chidamber and Kemerer (Chidamber and Kemerer, 1994) and Briand et al. (Briand et al., 1997). This subset, consisting of metrics that can be collected during the design stage of a project, includes inheritance and coupling metrics (and excludes cohesion and traditional complexity metrics). They are summarized in Table 1. Details of their computation and an illustrative Java example are given in the Appendix (Section 7).

Metric Acronym	Definition
NOC	This is the **Number of Children** inheritance metric (Chidamber and Kemerer, 1994). This metric counts the number of classes that inherit from a particular class (i.e., the number of classes in the inheritance tree down from a class).
DIT	The Depth of Inheritance Tree (Chidamber and Kemerer, 1994) metric is defined as the length of the longest path from the class to the root in the inheritance hierarchy.

ACAIC
OCAIC
DCAEC
OCAEC
ACMIC
OCMIC
DCMEC
OCMEC
	These coupling metrics are counts of interactions among classes. The metrics distinguish among the class relationships (friendship, inheritance, none), different types of interactions, and the locus of impact of the interaction (Briand et al., 1997).
	
	The acronyms for the metrics indicate what types of interactions are counted:
	• The first or first two letters indicate the relationship:
	  • **A**: coupling to ancestor classes;
	  • **D**: coupling to descendents; and
	  • **O**: other, (i.e., none of the above).
	• The next two letters indicate the type of interaction between classes c and d:
	  • **CA**: there is a class-attribute interaction between classes c and d if c has an attribute of type d; and
	  • **CM**: there is a class-method interaction between classes c and d if class c has a method with a parameter of type class d.
	• The last two letters indicate the locus of impact:
	  • **IC**: Import Coupling; and
	  • **EC**: Export Coupling

WMC	This is the **Weighted Methods per Class** metric (Chidamber and Kemerer, 1994), and can be classified as a traditional complexity metric. It is a count of the methods in a class. It has been suggested that neither methods from ancestor classes nor *friends* in C++ be counted (Basili et al., 1996; Chidamber and Kemerer, 1995). The developers of this metric leave the weighting scheme as an implementation decision (Chidamber and Kemerer, 1994). Some authors weight it using cyclomatic complexity (Li and Henry, 1993). However, others do not adopt a weighting scheme (Basili et al., 1996; Tang et al., 1999). In general, if cyclomatic complexity is used for weighting then WMC cannot be collected at early design stages. Alternatively, if no weighting scheme is used then WMC becomes simply a *size measure* (the number of methods implemented in a class), also known as **NM**.

Table 1: Summary of metrics used in our study. The top set are the coupling and inheritance metrics. The metric at the bottom is used to measure size.

2.3 Hypotheses

The above brief exposition has presented a cognitive theoretical basis for object-oriented metrics. We also presented the metrics that we are evaluating. Based on this we can state precise hypotheses about the direction and functional form of the association between each of the metrics and fault-proneness. In many instances there are competing theories. Table 2 includes a summary of all the hypotheses that we have formulated. The detailed justification for these hypotheses is presented in the Appendix (Section 8). The empirical study below tests these hypotheses and will allow us to determine which of the competing hypotheses can be supported empirically.

Export Coupling Hypotheses
• **H-EC+ : Positive Association** Classes with high export coupling have many assumptions made about their behavior. This makes it easier to violate some of these assumptions. This can be seen as a fan effect. Furthermore, more interference will occur as the engineers trace back frequently from classes with high export coupling to understand how they are used. Given that these mechanisms are in the same direction, it is not possible to disentangle them. We therefore consider them together as one hypothesis.
• **H-EC- : Negative Association** Classes with high export coupling are more familiar because they are consulted often. Furthemore, classes with high export coupling are given extra attention during development. Given that these two mechanisms are in the same direction, it is not possible to disentangle them. Therefore, by definition it is not possible in an observational study to determine which one of these two mechanisms is operating. We therefore consider them together as one hypothesis.
Inheritance Hypotheses
• **H-NOC0: No Association** There is no association between NOC and fault-proneness because the only effect of NOC will be due to the confounding influence of descendant-based export coupling.
• **H-DITQ: Qudratic Association** Classes at the root are consulted more often, and therefore they are more familiar. Classes deeper in the hierarchy are also consulted more often. Classes in the middle of the inheritance hierarchy are, in general, not consulted often and therefore they are not familiar. This suggests a quadratic relation between DIT and fault-proneness.
• **H-DIT+: Linear Positive Association** Root classes are most familiar, but the hierarchy is not well constructed, and therefore there exists higher conceptual entropy at the deepest classes. This will result in the root classes having few faults and the deeper classes having the most faults.
Import Coupling Hypotheses
• **H-IC+: Positive Association** Due to interference and fan effects, there will be a positive association between import coupling and fault-proneness.

Table 2: Summary of hypotheses relating the object-oriented metrics to fault-proneness.

3 Research Method

In this section we present the methodological details of our empirical study.

3.1 Measurement

3.1.1 Object-Oriented Metrics

The object-oriented design metrics described above were collected using a Java static code analyzer (Farnese et al., 1999). No Java inner classes were considered. In order to validate the metrics, we also need to control for the potential confounding effect of size (El-Emam et al., 2000b). The size measure used was NM (the total number of methods defined in the class).

3.1.2 Fault Measurement

In the context of building quantitative models of software faults, it has been argued that considering faults causing field failures is a more important question to address than faults found during testing (Binkley and Schach, 1998). In fact, it has been argued that it is the *ultimate* aim of quality modeling to identify post-release fault-proneness (Fenton and Neil, 1999). In at least one

study it was found that pre-release fault-proneness is not a good surrogate measure for post-release fault-proneness, the reason posited being that pre-release fault-proneness is a function of testing effort (Fenton and Ohlsson, 2000).

Therefore, faults counted for the system that we studied were due to field failures occuring during actual usage. For each class we characterized it as either faulty or not faulty. A faulty class had at least one fault detected during field operation. Distinct failures that are traced to the same fault are counted as a single fault.

3.2 Data Source

The system that was analyzed was a commercial Java application. This system is an XML document editor. It consists of a document browser that displays the document hierarchy, allowing for navigation within the document's XML structure, a property inspector for XML elements, a content editor with the basic features for rich text editing, and a rendition area for the document being edited. It had 145 classes, of which 34 had faults in them. In total, 170 post-release faults were fixed in these 34 classes.

3.3 Confounding

The above object-oriented metrics can be seen as capturing dependencies within the inheritance hierarchy and dependencies among classes. It is plausible that there is confounding between the two types of dependencies (in addition to size confounding). For example, consider the relationship between DIT and OCMIC for the C++ system in (El-Emam et al., 1999), as illustrated in Figure 2. Here, OCMIC is highest at the low DIT and lowest at high DIT. If there is an effect of DIT on fault-proneness, then this might be diluted due to the opposing effect of OCMIC. Such confounding effects are different for different systems. For example, consider the same relationship for the Java system in (El-Emam et al., 2001b), which is illustrated in Figure 3. Here, the exact opposite relationship is seen. The confounding effect would result in an inflated relationship for either DIT or OCMIC if the other metric is also related with fault-proneness.

Figure 2: Relationship between median OCMIC at each level of DIT for the C++ system in (El-Emam et al., 1999).

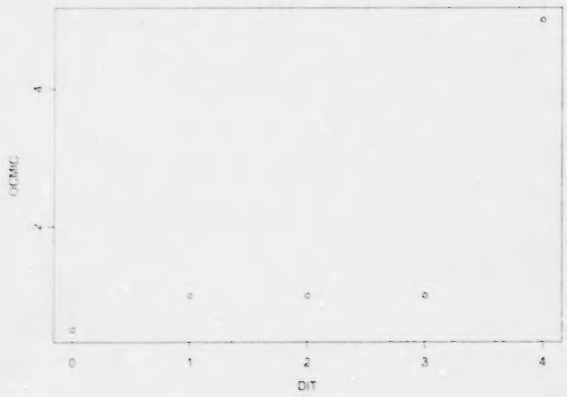


Figure 3: Relationship between median OCMIC at each level of DIT for the Java system in (El-Emam et al., 2001b).

Even seemingly predictable confounding effects may not be so. For instance, one study showed that classes lower in the inheritance hierarchy took more effort to modify because of the need to consult methods and attributes of parent classes (Prechelt et al., 1999). This suggests that DIT is confounded with Ancestor-based import coupling metrics. However, another study showed that Ancestor-based import coupling metrics do not load on the same component as DIT in a principal components analysis (Briand et al., 2000).

Therefore, during the analysis it is important to statistically control for such potential confounding effects. This means that if one wishes to validate a coupling metric (or test a coupling metric hypothesis), then it is necessary to control for depth of inheritance (DIT). Similarly, if one wishes to validate an inheritance metric, then it is necessary to control for the different types of coupling.

If one is validating a coupling metric and there is no confounding effect between the coupling metric and DIT, then this statistical control will not influence the conclusions drawn about the coupling metric. If there is a confounding effect, then the estimates for the coupling metric coefficient and conclusions drawn about its impact on fault-proneness will be more correct. Therefore, as a prudent measure, such potential confounders are controlled throughout our analysis.

3.4 Analysis Methods

The method that we use to perform our analysis is logistic regression. Logistic regression (LR) is used to construct models when the dependent variable is binary, as in our case. Logistic regression is used to validate the metrics and to construct the prediction model.

A summary of the analytical techniques that we used and that are described below is given in Table 3, as well as a mapping to the objectives of the analysis.

Objective	Analysis Method(s)
Validate Metrics/Test Hypotheses	Logistic Regression (Section 3.4.1)
Build Predictive Model	Logistic Regression (Section 3.4.2)
Evaluate Predictive Model	Evaluating Prediction Accuracy (Section 3.4.3) Evaluating Cost Savings (Section 3.4.4)

Table 3: The steps of our research method and the analytical techniques used.

3.4.1 Logistic Regression for Validating Individual Metrics

The general form of an LR model is:

$$\text{logit}(\pi) = \beta_0 + \sum_{i=1}^{k} \beta_i C_i + \beta_{k+1} M \qquad \text{Eqn. 1}$$

where π is the probability of a class having a fault, the C's are the confounding variables that need to be controlled, and M is the specific metric that we are evaluating. As noted above, we need to control for size, as measured using NM, and the depth of inheritance tree.

The β parameters are estimated through the maximization of a log-likelihood (Hosmer and Lemeshow, 1989). Testing the null hypothesis that a β parameter is equal to zero was done using a likelihood-ratio test (Hosmer and Lemeshow, 1989). All statistical testing was performed at an alpha level of 0.05. The specific functional forms that were tested are discussed further in the results.

We performed further diagnostics on the LR models, namely to evaluate the model, to test for collinearity, and to identify influential observations. The details of these methods have been presented in (El-Emam et al., 2001b). Important methodological points to summarize here are that we use the likelihood ratio statistic, G, to test the significance of the overall model, and the Hosmer and Lemeshow (Hosmer and Lemeshow, 1989) R^2 value as a measure of goodness of fit. Furthermore, we compute the condition number, η, to determine whether dependencies among the independent variables are affecting the stability of the model (Belsley et al., 1980).

The effect size of a metric is measured using the change in odds ratio (El-Emam et al., 2001b). It allows us to evaluate how large the effect of a particular metric on the probability of a fault, and compare magnitude of the effects of different metrics. For a variable x, the odds ratio is computed as $\Delta\Psi = e^{\beta\sigma}$, where β is the variable's estimated parameter, and σ is the standard deviation of x from our data. An odds ratio value above one indicates a positive relationship, and a value less than one indicates a negative relationship. The odds ratio is only applicable when the relationship between a metric and fault-proneness is monotonic. If the relationship is not monotonic, then there is no single $\Delta\Psi$ value for the effect of the metric.

3.4.2 Building Predictive Model

To make predictions about class fault-proneness, we construct a multivariate logistic regression model:

$$\text{logit}(\pi) = \beta_0 + \sum_{i=1}^{k} \beta_i C_i + \sum_{j=k+1}^{p} \beta_j M_j \qquad \text{Eqn. 2}$$

where now there are the confounding variables, C_i, and a set of object-oriented metrics denoted by M_j. In order to construct a predictive model, it is necessary to select a subset of metrics, M_j, to use. In principle, one can use a stepwise selection approach (see (Hosmer and Lemeshow, 1989) for a description of stepwise selection). Stepwise selection is an automated procedure for selecting the "best" subset of metrics to include in the predictive model based on repeated statistical tests. This approach has been used in a number of object-oriented metrics validation studies (Briand et al., 1998b, 1999b; Briand et al., 2000).

However, in general, stepwise selection procedures for regression analysis have been found not to produce believable results because of the repeated statistical tests that they use. A Monte Carlo simulation of stepwise selection indicated that in the presence of collinearity amongst the independent variables, the proportion of 'noise' variables that are selected can reach as high as 74% (Derksen and Keselman, 1992). It is clear that in object-oriented metrics validation studies many of the metrics are correlated (Briand et al., 1998b; Briand et al., 2000). Another Monte Carlo study done by Flack and Chang (Flack and Chang, 1986) confirms the fact that noise variables are often selected by these procedures. PA, the percentage of the repeated samples from which all authentic variables were selected, was low. For the stepwise procedure, PA was 34%, when there is a correlation among predictor variables of 0.3 and the best case with 40 observations and 10 candidate predictor variables. The median of P_N, the percentage of selected noise variables, was higher than 50% for most models, showing that for most of the samples at least half of the selected variables were noise. The median of P_N increased with the number of candidate predictor variables. Harrell and Lee (Harrell and Lee, 1984) note that when statistical significance is the sole criterion for including a variable the number of variables selected is a function of the sample size, and therefore tends to be unstable across studies. Furthermore, some general guidelines on the number of variables to consider in an automatic selection procedure given the number of 'faulty classes' are provided in (Harrell et al., 1996). The studies that used automatic selection (Briand et al., 1998b; Briand et al., 2000) had a much larger number of variables than these guidelines. Therefore, clearly the variables selected through such a procedure should not be construed as the best object-oriented metrics nor even as good predictors of fault-proneness.

Consequently, we opted for a manual procedure for selecting variables to include in the prediction model. This involves selecting the metrics that are statistically significant during the individual validation stage described above that are least correlated with each other. A similar procedure has been used in previous studies (El-Emam et al., 1999; El-Emam et al., 2001b).

3.4.3 Evaluating Prediction Accuracy

Below we describe how we evaluated the prediction accuracy of the multivariate prediction model.

Ideally, when evaluating a prediction model, one would like to have a data set for building the model and a separate data set for evaluating the prediction accuracy of the model. However, this requires large data sets. An alternative when data sets are not large is to use a leave-one-out cross-validation (Weiss and Kulikowski, 1991). In leave-one-out cross-validation the analyst removes one observation from the data set, builds a model with the remaining $n-1$ observations, and evaluates how well the model predicts the value of the observation that is removed. This process is repeated each time removing a different observation. Therefore, one builds and evaluates n models.

A logistic regression model makes probability predictions (values between zero and one). One can either dichotomize this probability prediction and evaluate the binary classification accuracy. However, the choice of an appropriate cutoff point is difficult. The approach that we use avoids dichotomization.

We use Receiver Operating Characteristic (ROC) curves. The rationale for using ROC curves is described in the appendix, and the types of problems it addresses in other accuracy measures

are discussed. The area under the ROC curve (referred to as AUC) is our measure of prediction accuracy. This gives the estimated probability that a randomly selected class with a fault will be assigned a higher predicted probability by the logistic regression model than another randomly selected class without a fault. Therefore, an area under the curve of say 0.8 means that a randomly selected faulty class has an estimated probability larger than a randomly selected not faulty class 80% of the time.

3.4.4 Evaluating Cost Savings

We wish to use the prediction model at the early stages to predict the classes that are fault-prone. Once identified, the project may take preventative actions for these classes, such as considering alternative design options for the fault-prone classes, or assign more experienced technical staff to work on the fault-prone classes (for example, during inspections and testing). Furthermore, any changes proposed for the fault-prone classes can entail a more stringent justification and inspection.

In this subsection we describe a model for calculating the potential cost savings from using the prediction model. The model is general, but we instantiate it with values that are specific to the organization with whom we performed this study. In formulating this model we will err on the conservative side. If the cost savings are still substantial with such conservatism, then one can argue that in practice the cost savings may be even higher.

In the context of our study, it was decided to inspect the design document for the fault-prone classes using Perspective-Based reading (PBR) techniques (Laitenberger et al., 2000). The organization currently does not have a formal inspection of its design documents. Therefore, classes that are considered to be fault-prone will be inspected. We use this example for the cost savings discussion, although any other fault detection technique could be used.

The premise of inspecting fault-prone classes before the product is released is that it will result in cost savings and more satisfied customers (because fewer faults will be discovered by the customers). We will focus on the former.

The costs of customers discovering faults can be high. A support organization must be set up to log the customer failure reports. These are then passed to the development organization where the failures have to be recreated. Recreating failures is costly because a system configuration similar to the customer's has to be set up as well. Once a failure is recreated the cause has to be identified. The appropriate classes with faults in them have to be fixed and retested. This involves running a regression test suite as well as developing new test cases for the particular failure discovered. A fix has to be sent to the customer to solve his/her immediate problem. Multiple fixes are then packaged together for a subsequent release. A minor release consists almost exclusively of fault fixes. This has to be shipped to all customer sites with instructions for the upgrades. This is a costly process that exists only because customers discover faults. Note that there may be other faults in the software that the customers do not discover. These undiscovered faults do not add to the organization's post-release costs.

		Predicted Fault Status		
		Not faulty	**Faulty**	
Real Fault Status	**Not faulty**	n_{11}	n_{12}	$N_{1.}$
	Faulty	n_{21}	n_{22}	$N_{2.}$
		$N_{.1}$	$N_{.2}$	N

Table 4: Confusion matrix showing notation for actual versus predicted faults.

We follow the notation given in Table 4 for formulating the cost savings. Let us say that the (average) cost of a customer found fault is C_c. Furthermore, let the cost of performing a PBR

inspection be C_{PBR}. If a prediction model is used to identify fault-prone classes, then the potential savings will be: $F_{22}C_C$, where F_{22} is the number of faults in the classes correctly identified as fault-prone in the n_{22} cell. The $F_{22}C_C$ value is the post-release cost that is avoided by performing an early inspection. However, this assumes that the inspection will find all the faults in the classes identified as fault-prone. This is unrealistic. The study in (Laitenberger et al., 2000) concluded that PBR inspections will, on average, find 58% of the faults in the inspected object-oriented design artifacts. Therefore, the post-release costs avoided can be given by $PF_{22}C_C$, where P is the proportion of faults found. It should be noted that the 58% value was obtained by subjects who had recently been trained on PBR. As experience accumulates one would expect this value to increase.

The prediction model predicts that N_{+2} classes will be faulty, and therefore all these classes would need to be inspected. The cost of inspections is given by $N_{+2}C_{PBR}$. The total cost savings is then given by:

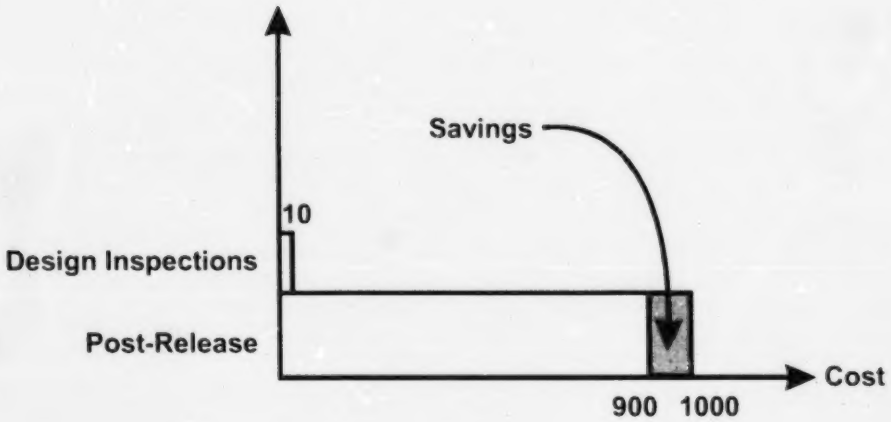
$$Cost\ Savings = PF_{22}C_C - N_{+2}C_{PBR} \qquad \textbf{Eqn. 3}$$

Khoshgoftaar et al. (Khoshgoftaar et al., 1998) define a Return on Investment (ROI) model as follows:

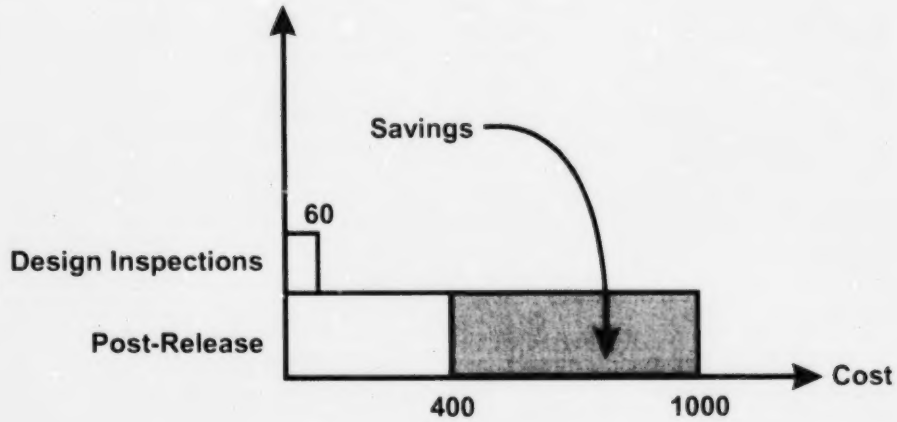
$$ROI = \frac{PF_{22}C_C - N_{+2}C_{PBR}}{N_{+2}C_{PBR}} \qquad \textbf{Eqn. 4}$$

This ROI model captures the costs saved beyond the costs invested. It is similar to the ROI model used at HP for evaluating the benefits of other fault detection techniques (Franz and Shih, December 1994).

However, Kusumoto (Kusumoto, 1993) noticed that his kind of model introduces a discrepancy when evaluating the benefits of a fault detection technique.



(a)



(b)

Figure 4: Examples for calculating return on investment.

Consider two prediction models, (a) and (b). In both cases assume that if a prediciton model was not used post-release fault discovery would cost 1000 units. Using prediction model (a) to decide on what to inspect consumes 10 units of inspection cost, and saves 100 units. This is depicted in panel (a) of Figure 4. Therefore, the total savings would be 100-10=90, and the ROI is 90/10=9. The total costs incurred using this model would be 910 units. The second qualty model, (b), saves 600 units and its application costs 60 units on inspections. This is depicted in panel (b) of Figure 4. Therefore, total savings are 600-60=540, and the ROI is 540/60=9. The total cost incurred using (b) is 460 units. Using model (b) results in much less post-release costs than model (a), but they both have the same ROI. Clearly one would prefer model (b) since it has cost savings that are considerably larger than using model (a).

To alleviate this, the application of Kusuomoto's model involves taking into account the virtual post-release cost, $F_{2+}C_C$, where F_{2+} is the total number of post-release faults in the system. This is the cost that would be incurred had no prediction model been used at all. The total cost savings are therefore:

$$Savings = \frac{PF_{22}C_C - N_{+2}C_{PBR}}{F_{2+}C_C} \qquad \textbf{Eqn. 5}$$

The interpretation of this model is straight forward. It gives the proportion of costs due to post-release faults that would be saved by implementing PBR inspections on the classes that were predicted to be faulty by the prediction model. Therefore, a value of say 20% means that 20% of the costs of post-release faults would be saved. It can be re-formulated as follows:

$$Savings = \frac{PF_{22}}{F_{2+}} - \frac{N_{+2}}{F_{2+}k} \qquad \textbf{Eqn. 6}$$

where $k = \frac{C_C}{C_{PBR}}$, which is the cost ratio of dealing with a single post-release fault to performing a PBR inspection. Now we must determine what is a reasonable value for k. A recent study, for example, notes that the cost of finding and fixing faults post-release can be 200 times higher than finding and fixing them pre-release (Khoshgoftaar et al., 1998). Therefore, the k value can be in some cases extremely large.

If an organization uses PBR as the early fault detection mechanism, then one study with professionals found that the average effort to perform a PBR inspection is 670 minutes[8] (Laitenberger et al., 2000). This excludes fixing the fault. Faults found during inspections are relatively easy to fix since the exact location and cause are identified during the inspection process. In our study, the project manager estimated that a cost ratio k of at least 4 was reasonable in that particular organization's context (as noted above, we have an estimate of the average cost of a PBR inspection, and this was used as the basis for that k value). This would provide a conservative estimate of the savings from using a prediction model.

In this cost savings model it is assumed that a class that is flagged as fault-prone will be inspected by itself. In practice, due to the dependencies among classes in an object-oriented design, multiple classes will be inspected together that perform a single functionality. Therefore, the cost of inspections in this model are exaggerated compared to what one would witness in practice. This means that our cost savings estimate will be conservative.

In order to use the prediction model, one can rank the classes by their predicted probability of a fault. From a previous project within the same organization it was known that 39% of classes have post-release faults (El-Emam et al., 2001b). This was for a similar product also in first release. Therefore, using that as the basis for a decision, a project manager can select the top 39% of classes in terms of predicted fault-proneness and target these for PBR inspections. Although we know in retrospect that the actual percentage of faulty classes in our product was less, when applying the prediction model the project manager would not know that. Therefore we use the value that the project manager would use to simulate a relatisic application of the prediction model.

---

[8] This result was an average for three person inspection teams. Faults were found individually during preparation, and a meeting was performed for fault collection.

4 Results

4.1 Descriptive Statistics

The descriptive statistics for the object-oriented metrics and the size measure for the Java data set are shown in Table 5. The table shows the mean, standard deviation, median, inter-quartile range (IQR), the minimum and maximum, and the number of observations that are not equal to zero. The set of metrics analyzed consists of two inheritance metrics, Depth of Inheritance Tree (DIT) and Number of Children (NOC) and eight coupling metrics, of which four are Import Coupling metrics (OCAIC, OCMIC, ACAIC, ACMIC) and four are Export Coupling metrics (OCAEC, OCMEC, DCAEC, DCMEC). A more detailed description of these metrics is given in the appendix.

Variables DCAEC and DCMEC have less than five observations that are non–zero. Therefore, we exclude them from further detailed analysis. The size metric we used was the number of methods in a class (NM).

	Mean	Median	Std. Dev.	IQR	Min.	Max.	NOBS ≠ 0
DIT	2.23	1	1.75	2	0	7	144
NOC	0.55	0	3.07	0	0	29	16
OCAEC	0.8	0	1.79	1	0	14	53
OCAIC	1.055	0	2.077	1	0	14	61
OCMEC	2.64	0	7.71	2	0	69	64
OCMIC	2.17	1	3.46	3	0	21	77
ACAIC	0.151	0	0.544	0	0	4	14
ACMIC	1.48	0	3.35	2	0	24	61
DCAEC	0.096	0	0.8	0	0	9	3
DCMEC	0.42	0	4.28	0	0	51	3
NM	12.23	7	13.85	15	0	83	143

Table 5: Descriptive statistics for all of the object-oriented metrics on the java data set.

4.2 Testing the NOC Hypotheses

We expect that there will be no relationship between NOC and fault-proneness. Furthermore, we noted that there is a confounding effect between NOC and descendant-based export coupling.

We saw from the descriptive statistics above that only three classes have descendant type export coupling. Therefore we cannot construct a model controlling for the effect of Descendant-based coupling. However, Figure 5 shows that the few classes with the largest NOC values will have greater descendant-based coupling.

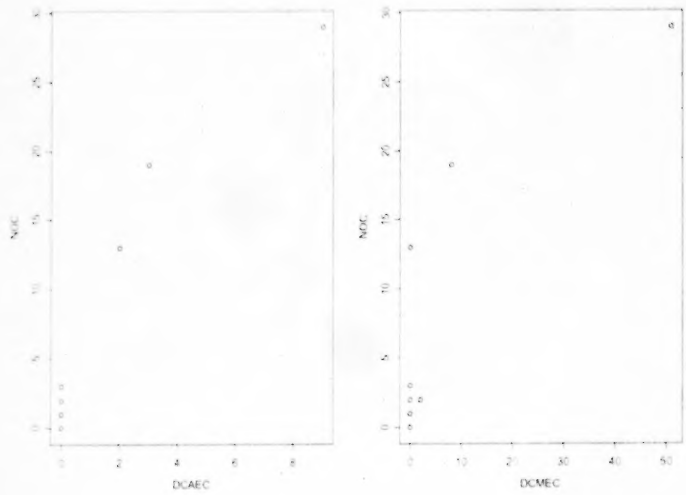


Figure 5: Relationship between descendant based export coupling and number of children.

The results of a formal test of hypothesis are presented in Table 6. These indicate that there is a statistically significant effect of NOC on fault-proneness. However, this includes the confounding effect of descendant-based export coupling. Since we cannot explicitly control for this type of export coupling because the metrics have a very low variation, we removed one observation with the largest value of descendant-based export coupling from the data set, this renders the NOC to fault-proneness association non-significant at an alpha level of 0.05. Therefore the positive association between NOC and fault-proneness is dependent on the fact that the largest NOC values are also the largest descendant-based export coupling values. This result is as expected.

G		R^2	
33.14 (2 d.f.); $p < 0.0001$		0.21	
	Intercept	**NM**	**NOC**
Coefficient	-2.41	0.08	0.309
p-value	<0.0001	<0.0001	0.049
$\Delta\Psi$		3.03	2.59

Table 6: Logistic regression results for testing the NOC hypotheses.

4.3 Testing the DIT Hypotheses

To test the DIT hypotheses we only need to construct two models, a linear and a quadratic one. Since these are nested models, they can be compared using a likelihood ratio statistic (see Eqn. 7 vs. Eqn. 8 below):

$$logit(\pi) = \beta_0 + \beta_1 NM + \beta_2 DIT \qquad \textbf{Eqn. 7}$$

$$logit(\pi) = \beta_0 + \beta_1 NM + \beta_2 DIT + \beta_3 DIT^2 \qquad \textbf{Eqn. 8}$$

The results for the DIT hypotheses are shown in Table 7. The parameter for the quadratic term is statistically significant.[9] Since this is based on a likelihood ratio test, it indicates that the quadratic model is preferred over the linear model. Therefore, the evidence supports the quadratic hypothesis. We do not show the change in odds ratio for the DIT metric because the change depends on the specific value of DIT, and hence no single universal $\Delta\Psi$ value for DIT makes sense. Figure 6 shows the relationship between DIT and fault-proneness. This indicates that the impact of DIT depends on the value of DIT (i.e., for values of DIT below the peak the change in odds ratio will be greater than one, and for values of DIT above the peak the change in odds ratio will be below one). This plot also shows that the results strongly support the familiarity mechanism postulated earlier.

G			**R^2**	
45.78 (3 d.f.); p<0.0001			0.289	
	Intercept	**NM**	**DIT**	**DIT^2**
Coefficient	-5.383	0.088	2.47	-0.352
p-value	<0.0001	<0.0001	0.00012	0.0005
$\Delta\Psi$		3.42	--	--

Table 7: Logistic regression results for testing the DIT hypotheses.

[9] The use of partial residual plots (Landwehr et al., 1984) also suggest strongly that a quadratic model is appropriate

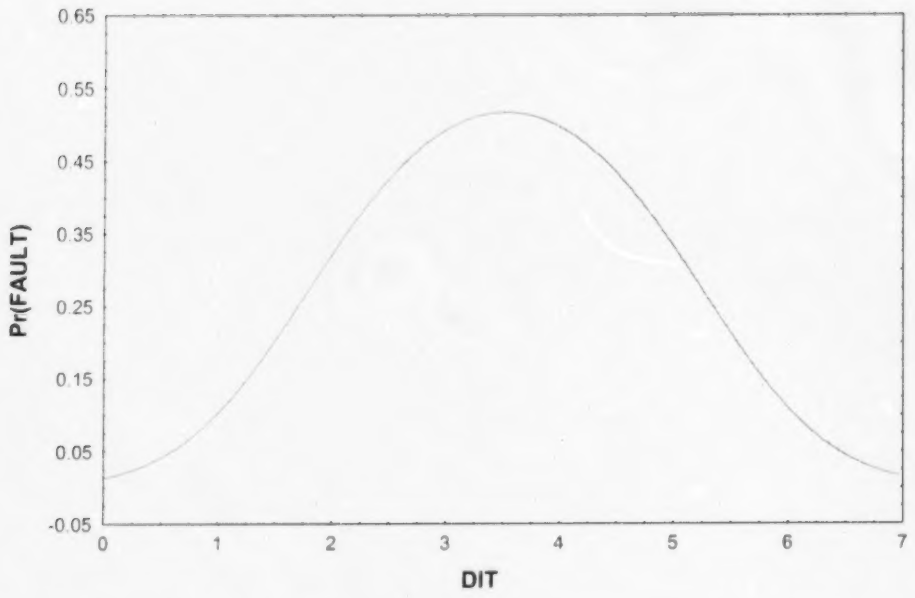


Figure 6: Relationship between DIT and the probability of a post-release fault at the mean value of NM.

4.4 Testing the Export Coupling Hypotheses

It was noted in Section 3.3 that inheritance metrics may be confounded with coupling metrics. Therefore, to test the export coupling hypotheses, we need to control for the potential confounding effect of inheritance depth. Recall that, if in this particular system there is no confounding effect, the inclusion of the inheritance depth variable in the model will not affect the estimates for the export coupling metrics. However, if there is a confounding effect then the parameter estimates will be more accurate.

We formulate the following two models for the export coupling metrics:

$$logit(\pi) = \beta_0 + \beta_1 NM + \beta_2 DIT + \beta_3 DIT^2 + \beta_4 OCAEC \qquad \textbf{Eqn. 9}$$

$$logit(\pi) = \beta_0 + \beta_1 NM + \beta_2 DIT + \beta_3 DIT^2 + \beta_4 OCMEC \qquad \textbf{Eqn. 10}$$

G			R^2		
50.87 (4 d.f.); p<0.0001			0.323		
	Intercept	**NM**	**DIT**	**DIT^2**	**OCAEC**
Coefficient	-5.78	0.091	2.62	-0.37	0.32
p-value	<0.0001	<0.0001	0.00013	0.0005	0.037
$\Delta\Psi$		3.5	--	--	1.59

Table 8: Logistic regression results for testing the OCAEC hypotheses.

G			R^2		
53.8 (4 d.f.); p<0.0001			0.347		
	Intercept	**NM**	**DIT**	**DIT^2**	**OCMEC**
Coefficient	-6.3	0.085	3.24	-0.49	0.077
p-value	<0.0001	<0.0001	0.000016	0.00006	0.039
$\Delta\Psi$		3.28	--	--	1.81

Table 9: Logistic regression results for testing the OCMEC hypotheses.

The results are shown in Table 8 and Table 9. They indicate that the export coupling metrics have a statistically significant association with post-release fault proneness. According to our hypotheses, these indicate that the familiarity hypothesis cannot be supported,[10] and that the interference and fan effects hypotheses receive empirical support.

4.5 Testing the Import Coupling Hypotheses

In a manner similar to the export coupling metrics, we formulate two models that control for the effect of inheritance depth:

$$logit(\pi) = \beta_0 + \beta_1 NM + \beta_2 DIT + \beta_3 DIT^2 + \beta_4 OCAIC \qquad \textbf{Eqn. 11}$$

$$logit(\pi) = \beta_0 + \beta_1 NM + \beta_2 DIT + \beta_3 DIT^2 + \beta_4 OCMIC \qquad \textbf{Eqn. 12}$$

[10] It is plausible that a familiarity effect exists, but its magnitude is small, and is cancelled by the competing positive effect. However, it is only possible to conjecture that this may be the case from this study.

G			R^2		
57.74 (4 d.f.); p<0.0001			0.37		
	Intercept	**NM**	**DIT**	**DIT^2**	**OCAIC**
Coefficient	-6.7	0.057	4.14	-0.72	0.47
p-value	<0.0001	0.016	<0.0001	<0.0001	0.0042
$\Delta\Psi$		2.19	--	--	2.68

Table 10: Logistic regression results for testing the OCAIC hypotheses.

G			R^2		
63.62			0.41		
	Intercept	**NM**	**DIT**	**DIT^2**	**OCMIC**
Coefficient	-6.03	0.053	2.38	-0.31	0.379
p-value	<0.0001	0.048	0.0004	0.002	<0.0001
$\Delta\Psi$		2.11	--	--	3.73

Table 11: Logistic regression results for testing the OCMIC hypotheses.

The results of testing these hypotheses are shown in Table 10 and Table 11. These indicate a strong positive and statistically significant relationship between these import coupling metrics and post-release fault-proneness.

Now we consider the ancestor-based import coupling. The ACAIC metric had only 14 non-zero values, and almost all of these were one. Thus this variable had restricted variation. The results for testing the ACAIC metric hypotheses are shown in Table 12. This indicates that there is no relationship between ACAIC and post-release fault-proneness.

G			R^2		
46.94 (4 d.f.); p<0.0001			0.298		
	Intercept	**NM**	**DIT**	**DIT^2**	**ACAIC**
Coefficient	-5.33	0.085	2.48	-0.36	0.45
p-value	<0.0001	<0.0001	0.0002	0.0007	0.37
$\Delta\Psi$		3.3	--	--	1.22

Table 12: Logistic regression results for testing the ACAIC hypotheses.

Figure 7 shows the relationship between ACMIC and DIT. It is clear that ACMIC is largest in the middle of the inheritance hierarchy. Given that we found the greatest fault-proneness in the middle of the inheritance hierarchy, this indicates a strong confounding effect of DIT on ACMIC. The same pattern was observed for ACAIC, although not as pronounced. This confounding explains why ACAIC was not significant, after controlling for the effect of depth of inheritance tree.

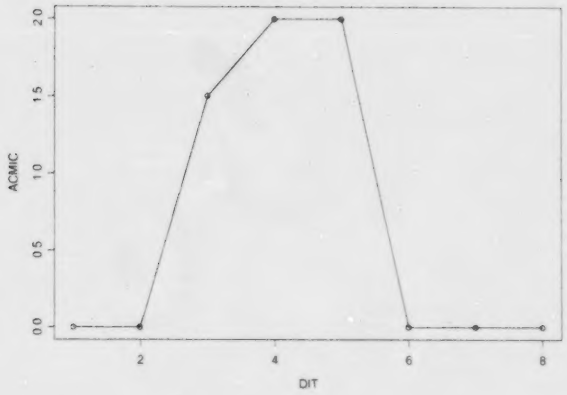


Figure 7: Relationtship between median ACMIC and DIT.

Similar conclusions can be drawn about ACMIC as seen in Table 13. The ACMIC metric is not associated with post-release fault-proneness after controlling for inheritance depth. It should also be noted that, when we *do not control* for inheritance depth we get highly significant ACAIC and ACMIC effects. Therefore, it is important to be cognizant of the confounding effect of DIT when building and interpreting validation models.

A post-hoc power analysis determined that the likelihood ratio test statistical power for ACAIC was 17% and for ACMIC was 9%. These are quite low values, indicating that the test was not sufficiently powerful at these sample and effect sizes to identify a significant effect if one existed.

G			R^2		
49.04 94 d.f.); p<0.0001			0.311		
	Intercept	**NM**	**DIT**	**DIT^2**	**ACMIC**
Coefficient	-5.433	0.088	2.45	-0.347	0.075
p-value	<0.0001	<0.0001	0.00058	0.0018	0.393
$\Delta\Psi$		3.39	--	--	1.23

Table 13: Logistic regression results for testing the ACMIC hypotheses.

4.6 Building the Predictive Model

As noted earlier, we use a manual approach for selecting the metrics to include in the predictive model. Table 14 shows the correlations among the significant coupling metrics. As can be seen the export coupling metrics are strongly correlated with each other and so are the two import coupling metrics. We therefore select one of each type that had the largest change in odds ratio in the validation models above. This leaves us with OCMIC and OCMEC.

	OCAEC	OCMEC	OCMIC	OCAIC
OCAEC	1.00			
OCMEC	0.54	1.00		
OCMIC	0.21	0.19	1.00	
OCAIC	0.31	0.30	0.51	1.00

Table 14: Correlations among the significant coupling metrics.

The final prediction model is shown in Table 15. The area under the ROC curve is 0.85 using a leave-one-out cross-validation. This can be considered a very good value indicating a high prediction accuracy.

G (p-value)			R^2		
65.7 (5 d.f.); p < 0.0001			0.42		
	NM	**DIT**	**DIT^2**	**OCMIC**	**OCMEC**
Coefficient	0.039	2.39	-0.318	0.3874	0.0826
p-value	0.185	0.00042	0.0023	<0.0001	0.016
$\Delta\Psi$	1.73	--	--	3.829	1.89

Table 15: Logistic regression results for the best model.

4.7 Summary of Hypothesis Tests

In general, our results support the cognitive theory. We also witnessed a confounding effect of depth of inheritance tree on some of the coupling metrics. Therefore, it is prudent to control for inheritance depth when validating coupling metrics. Consistent evidence was found for interference and fan effects.

The substantive results can be summarized as follows:

NOC Hypotheses: We expected that NOC would not be associated with fault-proneness after controlling for descendant-based export coupling. This was the results obtained, and is consistent with previous work.

DIT Hypotheses: Classes at the root of the hierarchy or that are deepest in the hierarchy are most familiar because they are consulted often. Therefore, due to the familiarity, they will have a low fault-proneness compared to class in the middle of the hierarchy.

Export Coupling Hypotheses: Classes with high export coupling have many assumptions made about their behavior. This can be considered a form of fan effect. Furthermore, more interference will occur as the engineers trace back frequently from classes with high export coupling to understand how they are used. This result is supportive of the cognitive hypothesis.

Import Coupling Hypotheses: Classes with high import coupling to other classes are more fault-prone due to interference and fan effects. Ancestor-based import coupling is not related with fault-pronensss. However, this result may be due to the low statistical power of the current study.

It is worthwhile to point out that most of our results are congruent with previous research. One exception is the quadratic relationship with DIT. There are two explanations for this. First,

previous studies did not look for a quadratic relationship.[11] Second, in previous studies DIT did not vary very much due to the lack of use of inheritance. This would explain why a quadratic relationship may not be found, even if the authors attempted to look for it.

Another exception is that some previous studies did not find a relationship between import coupling between classes and fault-proneness (El-Emam et al., 1999). This may be due to these studies not controlling for the effect of inheritance depth. Similarly, previous studies found an effect for ancestor-based import coupling and we did not. We noted a confounding effect of DIT and the ancestor-based import coupling metrics, and this was not accounted for in previous studies. This explains the difference in findings.

4.8 Potential Cost Savings from Using the Prediction Model

We used a leave-one-out technique to estimate the potential cost savings from using the prediction model. As noted earlier, this would give us a realistic estimate of the values in the confusion matrix should the prediction model be applied on a future system. We use Eqn. 8 to compute the cost savings.

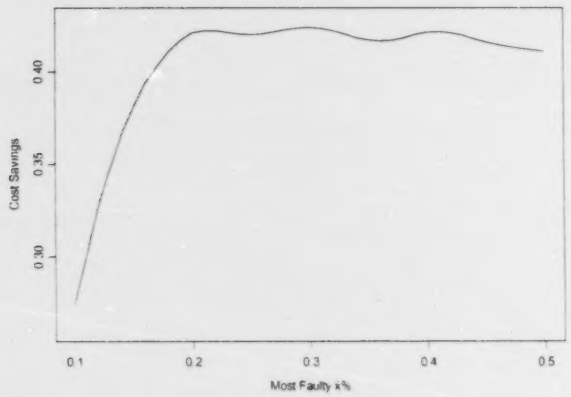


Figure 8: Cost savings from applying the prediction model by using PBR inspections on the top x% classes in terms of predicted fault-proneness for a cost ratio of 4. This is a smoothed plot.

A graph showing the cost savings is given in Figure 8. The x-axis shows the percentage of classes the project manager decides to inspect based on the predicted fault-proneness of the prediction model. For example, if the project manager had selected the top 39% of classes for inspections[12] using the prediction model, approximately 42% of total post-release costs would be saved. For this organization, this would translate to a significant amount of cost savings given that post-release costs can be quite large. A slightly higher cost saving would have been obtained had the project manager known that only 23% of the classes were faulty for this particular product. Furthermore, note that in our formulation and selection of values we have deliberately tended to be conservative so as not to exaggerate the benefits of the prediction model. Therefore, it is plausible that benefits would be higher.

In principle, the savings model that we have presented can be used to evaluate the benefits when applying other fault detection techniques or combinations of them. Our application used inspections with PBR reading since this particular technique was of interest to the organization.

---

[11] At least this was not mentioned in published studies.

[12] This is the value that the project manager would reasonably use based on a previous similar project by the same team (El-Emam et al., 2001b)

4.9 Discussion

We have articulated a cognitive theory of object-oriented metrics. The empirical study had two objectives: (a) to formally test this theory, and (b) build a useful post-release fault-proneness prediction model and evaluate its potential benefits in terms of cost savings.

4.9.1 Testing the Cognitive Theory

The cognitive theory was based on contemporary models of human memory. Three cognitive mechanisms were postulated. These are described below.

The first was that of familiarity. Object-oriented metrics that can be construed to measure familiarity were hypothesised to have a negative association with fault-proneness. We found that depth of inheritance tree (DIT) is a good measure of familiarity and, as predicted, it has a quadratic relationship with fault-proneness. We could not find evidence that familiarity would explain the impact of export coupling on fault-proneness. In fact, our results showed a positive relationship rather than the predicted negative relationship.

The pair of mechanisms of interference and fan effects were used to predict the impact of export and import coupling, and the evidence followed the predictions faithfully.

4.9.2 Prediction Model and Cost Savings

Based on the metrics that were validated, we then constructed a prediction model. The model could be used to predict which classes are faulty and for ranking the classes by their fault-proneness. Since the model can be constructed using design information only, it can trigger early fault detection activities, such as inspections or more test cases, for the high risk classes. We demonstrated that the cost savings from using such a model to focus inspections can be substantial, potentially reaching 42% savings of post-release costs for the organization with whom we performed this study.

Had the project manager decided to inspect all of the classes rather than the top 39% as indicated by the prediction model, then the cost savings are 37%. This indicates that the object-oriented metrics prediction model results in an additional cost savings of 5% compared to inspecting everything.

However, the cost savings are not the only business benefit of using the object-oriented metrics prediction model. In this particular project there was a fixed resource pool (i.e., the project team size was fixed). There was also a market window that the project had to meet. This is hardly a unique situation in commercial organizations. Using the prediction model will result in inspecting only 57 classes, whereas inspecting everything will result in inspecting 145 classes. According to our (conservative) assumptions, this means 57 versus 145 inspections.

Inspecting only the top 39% of the classes means spending approximately 636 person-hours on inspections before release[13]. Inspecting all classes means spending approximately 1619 person-hours of inspections before release. Therefore, by using the quality model approximately 983 person-hours of effort pre-release are also saved compared to inspecting all classes. For small projects with fixed resources, this amount of effort is not trivial. Thus, using the prediction model results in 5% more cost savings compared to inspecting all classes, and results in an approximately 61% saving in pre-release inspection effort compared to inspecting all classes.

Another potential benefit of focusing inspections is a reduced inspection interval. Votta (Votta, 1993) has shown that increased inspections result in increased interval if the inspection process includes a meeting. In practice, a meeting is held even if its purpose is only to log detected faults rather than to detect faults (see the discussion of this distinction in (Johnson and Tjahjono, 1998)). Given the shorter release times in today's business environment, increased interval can be quite detrimental to a product's commercial success. In principle, then, performing 57

[13] Recall that we have an estimate from a previous study that each inspection takes on average 670 minutes of effort (Laitenberger et al., 2000). These numbers exclude fix effort. However, even if fix effort is included, the conclusion does not change.

inspections compared to 145 inspections would reduce development interval. In our current study, however, we cannot quantify any potential benefits in terms of interval.

We must also qualify our discussion of the benefits of focused inspections. A recent study by Siy and Votta (Siy and Votta, 1999) argues that "minor faults" (which were mainly style and documentation issues) facilitate the evolution of code, and therefore result in reduced maintenance costs. While our cost-benefit analysis was concerned with design inspections, one can argue that similar style and documentation issues are also relevant, for example, in choosing variable names and in documenting high-level pseudo-code in design documents. If indeed there are maintenance cost savings from detecting and fixing such "minor faults", then it becomes more important to inspect all classes since the long-term ramifications may be nontrivial.

4.9.3 Confounding Effects

In constructing the models in this work, we paid special attention to confounding effects. A primary confounder for object-oriented metrics is size (El-Emam et al., 2000b). We also noted a confounding effect of inheritance depth. The structure and behavior of classes depends on where they are in the inheritance hierarchy. For instance, we found that classes in the middle of the hierarchy tended to exhibit more ancestor-based import coupling than at either end. Such confounding effects and their strength will enevitably depend on the design of the system being studied. This makes it important to inspect the relationship of a system's inheritance hierarchy with other metrics carefully during a validation study in order to understand where potential confounding effects may be. Not all systems are designed the same. Accounting for confounding effects may make the difference between finding a metric to be empirically valid or not. An implication of this is that without careful consideration of confounding, different studies will produce different results. A lack of consistent results will not make for a convincing case to adopt object-oriented metrics in practice. Furthermore, accumulation of scientific knowledge about object-oriented metrics will be retarded by a plethora of studies showing inconsistent findings. It is only hoped that accounting for such confounding effects will lead to consistent results across studies.

4.10 Limitations

Our study was driven by the cognitive theory that was formulated in Section 2. As seen from the results, by using a theory-driven approach to the validation of our metrics, we gained important insights that would not have been noted in an atheoretical study.

However, it must be recognized that the above cognitive theory suggests only one possible mechanism of what would impact external metrics. Other mechanisms can play an important role as well.

For example, some studies showed that software engineers experiencing high levels of mental and physical stress tend to produce more faults (Furuyama et al., 1994, 1997). Mental stress may be induced by reducing schedules and changes in requirements. Physical stress may be a temporary illness, such as a cold. Therefore, cognitive complexity due to structural properties, as measured by product metrics, *can never be the reason for all faults*.

If stress is not associated with structural properties then there is no confounding in our study and its results. The parameter estimates and significance tests that we obtained *remain valid*. But, if stress is associated with structural properties, then there is potential confounding. Consider the scenario where a core functionality is being developed by a group of engineers, and the remainder of the product development depends on this core functionality being available. Further, assume that the core classes will have high export coupling because the remaining classes will use their services extensively. One can then argue that the group of engineers developing the core functionality are under pressure to complete their part, and hence will introduce more faults. These faults will be in the classes with the highest export coupling. This would be a confounding effect on export coupling. Although the above scenario is plausible in principle, for our particular study the core functionality was provided by an external library and therefore it is highly unlikely to be reflective of the situation in the project from which we collected data.

Another potential confounding effect is developer experience. There exists evidence that experts are better than novices at constructing correct mental representations of object-oriented programs (Burkhardt et al., 1999). This is not surprising. However, one can imagine scenarios where the best developers are assigned to work on the classes with say large coupling. If this happens then there is a confounding effect of expertise and the structural properties that we wish to measure. If this was the case in our study, then we would expect to see a negative relationship between the coupling metrics and fault-proneness. That this was not the case is evidence that no such deliberate assignments were made. This was further confirmed by the project manager.

Another potential limitation of our study concerns the cost-benefit analysis. While the estimates that we have presented in this paper are pertinent to the organization we worked with, they will not necessarily be the same numbers obtained by other organizations. However, we did present the details of our methodology so that others can apply it to their context.

5 Conclusions

In this paper we presented a detailed cognitive theory to justify a number of object-oriented coupling and inheritance metrics. Based on this theory we generated a number of hypotheses relating the metrics to class fault-proneness. We then performed an empirical study to test these hypotheses. The study was performed on a commercial Java application. As part of this study we formulated a model to evaluate the cost-benefit of using object-oriented metrics prediction models, and applied to our results.

Our results indicate that the Depth of Inheritance Tree (DIT) is a good measure of the cognitive concept of "familiarity", and that it has a quadratic relationship with fault–proneness. We found that the Number of Children (NOC) metric was not associated with fault-proneness after controlling for the confounding effect of the Descendent based export Coupling metrics. Export Coupling metrics were found positively associated with fault-proneness, due to the interference and fan effects mechanisms. Therefore, the hypotheses for NOC and Export Coupling metrics were confirmed.

The cognitive mechanisms of interference and fan effects are also confirmed for Import Coupling to other classes metrics, which were found to have a positive association with fault–proneness. Ancestor based Import Coupling metrics were not related to fault-proness when controlling for the confounding effect of DIT.

In constructing the models, we found that besides size, which is a primary confounder for object–oriented metrics, the Depth of Inheritance Tree (DIT) metric has a strong confounding effect on the validity of object–oriented metrics.

The prediction model constructed showed good accuracy. Our cost savings evaluation showed that there is a 42% reduction in post-release costs if the prediction model we build is used to select the classes to inspect.

In closing, it is necessary to recognize that the formulation of theories is important for a discipline. The theories explain phenomena that we observe (i.e., they provide the mechanism). Knowledge of mechanisms can potentially lead to rapid advances in the field. Given the dearth of prior theories in software engineering, it is a necessary to articulate a theory and put it out for criticism and empirical test. Some of these theories will not survive, but others will. We therefore need to continuously propose theories, explanations, and empirically test them.

In this work we have postulated a theory and tested it. Further studies should investigate the versimilitude of its predictions, and possibly expand or modify the theory.

6 Acknowledgements

We wish to thank Janice Singer and Norm Vinson for their valuable assistance in formulating the cognitive theory, and for their reviews of an earlier version of this paper. We appreciate the comments and feedback of Anatol Kark on drafts of this paper. Also, our thanks to the participants at the 2000 PSM User Group Conference who attended a workshop on object-oriented measurement; the discussions resulted in further clarifications and extensions to the current work.

7 Appendix A: Metrics Evaluated

The metrics used in our study are a subset of the two sets of metrics defined by Chidamber and Kemerer (Chidamber and Kemerer, 1994) and Briand et al. (Briand et al., 1997). This subset, consisting of metrics that can be collected during the design stage of a project.

At the design stage it is common to have defined the classes, the inheritance hierarchy showing the parent-child relationships among the classes, identified the methods and their parameters for each class, and the attributes within each class and their types. Detailed information that is commonly available within the definition of a method, for example, which methods from other classes are invoked, would not be available at design time. Therefore, metrics that require knowledge of method calling sequence and which class attributes are accessed from within methods cannot be computed. This then excludes some coupling metrics (such as CBO and RFC (Chidamber and Kemerer, 1994)) and cohesion metrics (such as LCOM (Chidamber and Kemerer, 1994)). This leaves us with a total of 10 design metrics that can be collected, two defined by Chidamber and Kemerer (Chidamber and Kemerer, 1994), and eight by Briand et al. (Briand et al., 1997).

7.1 Inheritance Metrics

The two metrics defined by Chidamber and Kemerer (Chidamber and Kemerer, 1994) and used in this study are summarized below.

7.1.1 DIT

The Depth of Inheritance Tree metric is defined as the length of the longest path from the class to the root in the inheritance hierarchy.

7.1.2 NOC

This is the Number of Children inheritance metric. It counts the number of classes which inherit from a particular class (i.e., the number of classes in the inheritance tree down from a class).

7.2 Coupling Metrics

Briand et al defined coupling metrics which are counts of interactions between classes. The acronyms for the metrics indicate what types of interactions are counted. We define below the acronyms and their meaning, and then summarize the design metrics used.

- The first two letters indicate the relationship (A: coupling to ancestor classes; D: Descendents; and O: other, i.e., none of the above). Although the Briand et al. metrics suite covers it, friendship is not applicable in our case since the language used for the system we analyze is Java.
- The next two letters indicate the type of interaction between classes c and d (CA: there is a class-attribute interaction between classes c and d if c has an attribute of type d; and CM: there is a class-method interaction between classes c and d if class c has a method with a parameter of type class d). Method-method interactions are not available at design time. There is a method-method interaction between classes c and d if c invokes a method of d, or if a method of class d is passed as parameter to a method of class c.
- The last two letters indicate the locus of impact (IC: Import Coupling; and EC: Export Coupling). A class c exhibits import coupling if it is the using class (i.e., client in a client-server relationship), while it exhibits export coupling if is the used class (i.e., the server in a client-server relationship).

7.2.1 OCAIC

The OCAIC metric is defined as:

$$OCAIC(c) = \sum_{d \in Others(c)} CA(c,d) \qquad \textbf{Eqn. 13}$$

where :

$$Others(c) = C - (Ancestors(c) \cup Descendents(c) \cup \{c\}) \qquad \textbf{Eqn. 14}$$

And C is the set of classes in the system. This counts CA–interactions from class c to classes that are not ancestors of class c.

7.2.2 OCAEC

The OCAEC metric is defined as:

$$OCAEC(c) = \sum_{d \in Others(c)} CA(d,c) \qquad \textbf{Eqn. 15}$$

This counts CA–interactions to class c from classes that are not descendants of class c.

7.2.3 OCMIC

The OCMIC metric is defined as:

$$OCMIC(c) = \sum_{d \in Others(c)} CM(c,d) \qquad \textbf{Eqn. 16}$$

This counts CM–interactions from class c to classes that are not ancestors of class c.

7.2.4 OCMEC

The OCMEC metric is defined as:

$$OCMEC(c) = \sum_{d \in Others(c)} CM(d,c) \qquad \textbf{Eqn. 17}$$

This counts CM – interactions from class c to classes that are not descendants of class c.

7.2.5 ACAIC

The ACAIC metric is defined as:

$$ACAIC(c) = \sum_{d \in Ancestors(c)} CA(c,d) \qquad \text{Eqn. 18}$$

This counts CA – interactions from class c to ancestors of c.

7.2.6 ACMIC

The ACMIC metric is defined as:

$$ACMIC(c) = \sum_{d \in Ancestors(c)} CM(c,d) \qquad \text{Eqn. 19}$$

This counts CM – interactions from class c to ancestors of c.

7.2.7 DCAEC

The DCAEC metric is defined as:

$$DCAEC(c) = \sum_{d \in Descendents(c)} CA(d,c) \qquad \text{Eqn. 20}$$

This counts CA – interactions from descendents of class c to class c.

7.2.8 DCMEC

The DCMEC metric is defined as:

$$DCMEC(c) = \sum_{d \in Descendents(c)} CM(d,c) \qquad \text{Eqn. 21}$$

This counts CM – interactions from descendents of class c to class c.

7.3 Example of Calculating the Metrics

In this subsection we present a simple Java example of calculating the above metrics. The examples should clarify exactly what is counted when computing the inheritance and coupling metrics.

We suppose that classes A and B are descendents of class D and that between the classes A, B and C and between C and D there is an "Other" type of relationship (i.e. other than Ascendent or Descendent).

7.3.1 Example of Class-Attribute (CA) interaction

From the Figure 9 we can notice the CA interaction between classes A and B through the attribute public_ab1. This attribute is of type of class A and is declared in class B, such as, if class A is changed, class B will be changed via the public attribute of class B.

This means that OCAIC (B) = 1 (Class B has an attribute of type A, public_ab1, while classes A and B have a relationship of type "Other").

In a similar way we can measure the following CA metrics:

ACAIC(B) = 1 (class B has an attribute of type D, public_ab2, while class D is an ancestor of class B).

OCAIC (C) = 2 (Class C has an attribute of type of class A, public_ab3, and one of type of class D, public_ab4, and A and D are in a relationship of type "Other" with class C).

OCAEC(D)=1 (class C has an attribute of type D, while C has a relationship of type "Other" with class D).

DCAEC(D) = 1 (class B, which is a descendant of class D, has an attribute of type of class D).

OCAEC(A) = 2 (two classes, B and C, have attributes of type A, and they have a relationship with A of type "Other").

7.3.2 Example of Class-Method (CM) interaction

There is a Class-Method interaction between class A and mb1, a method of class B, and between class A and mb2, a method of class C.

OCMIC(B) = 1 (class B has a method, mb1, which has a parameter of type of class A, and between B and A there is a relationship of type "Other")

OCMIC(C) =1 (class C has a method, mb2, which has a parameter of type of class A, and between C and A there is a relationship of type "Other")

OCMEC(A) = 2 (2 classes, B and C, have methods with parameters of type of class A)

7.3.3 Example of Inheritance metrics:

DIT(A) and DIT(B) are equal to 1, because they are descendents of class D.

All the other classes have a DIT= 0.

Class D has two descendents, A and B, such that NOC(D) = 2. All the other classes in the example have NOC=0.

```
Class D
{
        public int x;
        public int y;
        ....
}
class A extends D
{
          public int aa;
          public void ma1()
          { ...
          }
}
class B extends D
{
          public A public_ab1;
          public D public_ab2;
          private int i;
          private float r;
         ...
         private void mb1(A a1)
          {
                    A a2;
                  C a3;  ...
                   a2=a3.mb2(a1);
           }
}
Class C
{
         public A public_ab3;
         public D public_ab4;
          .........
         private A mb2(A a)
         {          ........
                    a.ma1();
                    return a;
           }
}
```

Figure 10: Example of calculating the metrics for sample Java code.

8 Appendix B: Details of Cognitive Theory

In this appendix we present the details of the cognitive theory, and present the rationale for the hypotheses that we test in our study. The basic premise of this theory is that comprehension affects fault-proneness of classes. Therefore, below we explain how measures of dependencies in object-oriented systems can have an impact on comprehension. The strategy we use is to start from comprehension and work back to the metrics. The path follows the constructs in Figure 1.

The presentation of the theory is detailed because it attempts to integrate research on program comprehension, models of human memory, and metrics of object-oriented structure.

8.1 Comprehension

8.1.1 Mental Models

For procedural programs, two comprehension strategies have been identified. These are described below and their consequences on mental model construction are explained.

The first strategy is *top-down* (Brooks, 1983; Koenemann and Robertson, 1991; Soloway and Ehrlich, 1984). For instance, Soloway and Ehrlich's theory (Soloway and Ehrlich, 1984) considers comprehension as a plan-based hypothesis-driven activity. Plans are schematic knowledge about how to carry out stereotypical actions in a program. There are different levels of plans. *Strategic* plans are global plans for the solution of the problem; *tactical* plans are (programming) language independent specifications of algorithms for solutions of specific parts of a problem; and *implementation* plans are plans for carrying out a tactical plan in a given programming language. In addition, *rules of discourse* are programming conventions that govern how plans are expressed and combined.

According to Soloway and Ehrlich's top-down theory, program comprehension starts with the programmer hypothesizing a high-level goal, then decomposing that into more specific sub-goals that should exist in the program having the given high-level goal. Having identified the goals and sub-goals, the programmer must then search through the program to determine whether these goals are satisfied.

The second strategy is *bottom-up*. The basis of work on bottom-up comprehension strategies comes from modern theories of text comprehension. These theories postulate that there are three levels of mental representation for text (Kintsch, 1986; Schmalhofer and Glavanov, 1986; van Dijk and Kintsch, 1983):

- *Verbatim representation*, which corresponds to the literal form of the text.
- *Propositional textbase*, which consists of the propositions in the text and their relationships.
- *Situation model*, which represents the situation in the world which the text describes.

Pennington (Pennington, 1987a, 1987b) applied this theory to the comprehension of procedural progams. In Pennington's mental model, in addition to the verbatim level, there are the *program* and *domain* models.

The program model corresponds to the propositional textbase in the text comprehension theory. Operations and control flow constitute the program model. Operations usually correspond to one line of code. Control flow is the temporal order of execution of operations in a program, including simple sequential control, looping, branching, and function calls.

Once the program model is developed by the comprehender, a situation model is constructed. Building the situation model is therefore based on further abstractions from the program model.

The domain model corresponds to the situation model in the text comprehension theory. Function and dataflow knowledge constitute the domain model. Function needs to be inferred from the program text in conjunction with the programmer's knowledge of the application domain. Part of this knowledge are the plans mentioned above, which help inferring functions. Dataflow concerns

the transformation of data elements. Unlike function information, dataflow information is explicitly represented in the program text; however, it is not highlighted by the structure of the program as is control flow, and therefore extracting dataflow information still exerts a cognitive burden.

An integration of these mental models into a single comprehension model has been presented by von Mayrhauser and Vans (von Mayrhauser and Vans, 1995a, 1995b). This integrated model consists of a *top-down* model, which captures the different type of plans and rules of discourse mentioned above, as well as Pennington's program and domain (called situation model by von Mayrhaiser and Vans) models.

The manner in which the different mental model components are constructed during comprehension depends on the expertise of the comprehender. Shaft and Vessey (Shaft and Vessey, 1995) make a distinction between two types of knowledge that make up expertise: programming knowledge (consisting of programming language and platform knowledge) and application domain knowledge. They performed a study where they demonstrated that when programmers are familiar with the domain they are able to determine the function of the program and its modules without having to examine the program in detail. Thus, domain experts follow a top-down strategy. When there is domain unfamiliarity, it is necessary to examine the detailed code of the program to comprehend it, and this leads to an inference of the purpose of the program and its modules. This is a bottom-up strategy. These results then suggest that domain knowledge is a key factor influencing mental model construction: it serves as a "motherboard" into which the comprehender "plugs" specific program knowledge (von Mayrhauser and Vans, 1997).

In comprehension studies involving large scale software, it was found that subjects followed a mix of both top-down and bottom-up strategies, frequently switching between the two strategies (von Mayrhauser and Vans, 1996, 1997; von Mayrhauser et al., 1997). Programmers will not have equal knowledge about all parts of a program. Therefore, they will follow a top-down strategy for the parts where they have extensive domain knowledge, and switch to a bottom-up strategy for parts where their domain knowledge is lacking. Furthermore, for large projects, even if the programmer had little domain expertise, they would simply not have the memory capacity to construct a full program model first (as suggested by Pennington's theory) before abstracting to a situation model. This was evidenced in one study (von Mayrhauser and Vans, 1996) where the number of rereferences to the top-down model increased with program size (for the program being comprehended), and references to the program model decreased with program size. Therefore, for real applications, switching among the three mental representations is necessary. For example[14], if while constructing the program model the programmer notices a clue indicating that a sort is being performed. The programmer then generates sub-goals to support the hypothesis and searches the code for clues to support these sub-goals. If during the search a section of unrecognized code is encountered, the programmer then jumps back to building the program model.

Another expertise factor that was found to be important was familiarity with the program being comprehended. Lack of familiarity resulted in the comprehender switching to the program model in order to understand it, even if s/he had application domain knowledge.

8.1.2 Mental Models of Object-Oriented Programs

The above studies were performed largely with procedural applications. However, recently a number of studies have tested elements of the above mental model structure on object-oriented software. The evidence supports the existence of a separate situation and program model (Burkhardt et al., 1997; Burkhardt et al., 1999; Corritore and Wiedenbeck, 1999, 2000).

Important structural components of the situation model are (Burkhardt et al., 1999; Corritore and Wiedenbeck, 1999): inheritance and composition relationships, client-server relationships between objects, and dataflow relationships within and among objects. Important structural components of the program model are: control flow and elementary functions (e.g., methods).

---

[14] This example is taken from (von Mayrhauser and Vans, 1997).

During comprehension it is necessary to search through the program in order to construct the program and situation model, and make connections amongst the three mental models. Below we argue that recall plays an important role in this process.

8.2 Relationship Between Comprehension and Recall

There exists an accumulation of evidence relating comprehension and recall. In one study, subjects were asked to read or listen to a number of unrelated sentences, and after that they were requested to recall the last word of each sentence (Daneman and Carpenter, 1980). The largest group of sentences for which they could recall the last word was defined as the reading or listening span. It was found that spans were strongly related to the subjects' comprehension scores. The authors argue that a larger reading and listening span indicates the ability to store a large portion of the text during comprehension. Similar evidence linking memory capacity and comprehension was presented in (Just and Carpenter, 1992). This suggests a direct link between recall and comprehension.

Schneiderman (Schneiderman, 1977) proposed that performance on a recall task would be a good measure of program comprehension. He demonstrates that in one experiment recall scores were strongly correlated with modifiability of programs. He then suggests that one way of assessing the ease of comprehension and modifiability of a program is to test its ease of memorization. In a second experiment he found a strong correlation between recall scores and scores on a comprehension test (Schneiderman, 1977). Hence, we can postulate that there is covariation (or an association) between recall and comprehension.

In a series of studies investigating information needs during program comprehension, von Mayrhauser and Vans (von Mayrhauser and Vans, 1995b, 1997; von Mayrhauser et al., 1997) found that information already browsed (e.g., list of variables) during comprehension was frequently required. This suggests that the ability to recall this information would play an important role in the ability to build the mental models.

Rist (Rist, 1996) suggested that in object-oriented systems, plans (in the top-down model) and progam objects are orthogonal. This means that a plan can use many objects and an object can be used in many plans. This delocalization of plans increases the difficulty of matching information in the top-down model to the situation and program models. But it also requires storing and recalling information about objects already browsed.

When subjects are not able to recall a particular fact, they retrieve related facts and make inferences (Anderson, 1995). However, inferences may lead to incorrect recall. Making such inferential errors has been demonstrated in a number of studies (Bransford et al., 1972; Sulin and Dooling, 1974), and these errors increase as more time elapses between recall and the time material was learned (Dooling and Christiaansen, 1977; Spiro, 1977). This, then, can lead to an inability to correctly construct the mental models and hence comprehension errors.

Below we present contemporary models of human memory. These models will allow us to determine which object-oriented metrics are expected to influence recall of information.

8.3 Recall

A model of human memory is important when considering recall. In the software engineering literature, this memory model is believed to consist mainly of short-term and long-term memory (STM and LTM respectively) (Brooks, 1999; Hatton, 1998; Tracz, 1979; von Mayrhauser and Vans, 1995a, 1995b).[15]

In this sub-section, we first elaborate and clarify modern theories of human memory, then identify the factors that affect recall, and state how these are related to dependencies among classes. Based on the mapping we present, we make precise predictions about the relationship between

---

[15] Tracz (Tracz, 1979) also discusses very-short-term memory, which plays a role in attention and perception. However, this does not play a big role in cognitive theories that are used to associate software product metrics to understandability.

the object-oriented metrics and fault-proneness. We also state competing hypotheses that may explain the same effect or that suggest opposite effects.

8.3.1 Object-Oriented Chunks

Cant et al. (Cant et al., 1995) argue that program comprehension consists of both *chunking* and *tracing*. Chunking involves recognizing groups of statements and extracting from them information, which is remembered as a single mental abstraction. These chunks are further grouped together into larger chunks forming a hierarchical structure. Tracing involves scanning through a program, either forwards or backwards, in order to identify relevant chunks. Subsequently, they formulate a model of cognitive complexity for a particular chunk, say D, which is the sum of three components: (1) the difficulty of understanding the chunk itself; (2) the difficulty of understanding all the other chunks upon which D depends; and (3) the difficulty of tracing the dependencies on the chunks upon which D depends. Davis (Davis, 1984) presents a similar argument where he states "Any model of program complexity based on chunking should account for the complexity of the chunks themselves and also the complexity of their relationship."

In order to operationalize this model, it is necessary to define a *chunk*. Tracz considers a 'module' to be a chunk (Tracz, 1979). However, it is not clear what exactly a module is. Cant et al. (Cant et al., 1995) make a distinction between elementary and compound chunks. Elementary chunks consist only of sequentially self-contained statements. Compound chunks are those which contain within them other chunks. Procedures containing a number of procedure calls are considered as compound chunks. At the same time, procedures containing no procedure calls may also be compound chunks. If a procedure contains more than one recognizable subunit, it is equivalent to a module containing many procedure calls in the sense that both contain within them multiple subchunks. Subsequent work by Cant et al. (Cant et al., 1994) operationally defined a chunk within object-oriented software as a method. However, Henderson-Sellers (Henderson-Sellers, 1996) notes that a class is also an important type of (compound) chunk.

8.3.2 Memory Span and STM

The classical theory of STM proposes that information went into an intermediate storage where it had to be rehearsed before it can go into the relatively permanent LTM (Anderson, 1995). If an item left STM before a permanent LTM representation was developed, it would be lost forever. One could not keep information in STM forever since new information would always be coming in and push the old information out. Essentially, information had to "do time" in STM before it can get to LTM. Furthermore, it was believed that STM had limited capacity.

In a software engineering context, Hatton (Hatton, 1997) argues that Miller (Miller, 1957) shows that humans can cope with around 7±2 pieces of information (or chunks) at a time in STM, independent of information content. He then refers to the text of Hilgard et al. (Hilgard et al., 1971). Hilgard et al. note that the contents of long-term memory are in a coded form and the recovery codes may get scrambled under some conditions. Short-term memory incorporates a rehearsal buffer that continuously refreshes itself. Hatton suggests that anything that can fit into short-term memory is easier to understand and less fault-prone. Pieces that are too large or too complex overflow, involving use of the more error-prone recovery code mechanism used for long-term storage. Tracz (Tracz, 1979) argues that STM has a fixed size, and therefore software complexity must not exceed the number of items capable of being held in STM at one time, and that the size of these items should be small enough and well formed enough to be handled by STM. He then also refers to the 7±2 limitation of STM. Hatton uses this reasoning in explaining complexity for procedural (Hatton, 1997) and object-oriented applications (Hatton, 1998).

It must be recalled that Miller's landmark article (Miller, 1957) on memory span is approximately 45 years old. Curtis (Curtis, 1979) notes that the 7±2 numbers describe cognitive processes related to simple stimuli rather than the complex information processing tasks involved in programming. He then states that "computer scientists would do well to immediately purge from their memories the magic number 7±2 ... These numbers have been incorrectly applied in too

many explanations and are too frequently cited by people who have never read the original articles." Memory span studies typically reported the number of items that can be recalled on 50% of occasions (Broadbent, 1975). This can hardly be considered a challenging test of recall. The actual number of items that can be recalled reliably is, however, much smaller than seven; closer to three (Broadbent, 1975). Furthermore, Miller noted that the span for letters is nearly the same as that for digits[16], and that for binary digits is a little more. Yet if subjects were trained to encode binary symbols into decimal digits they could reproduce after one presentation a long string of binary digits corresponding to about the same number of decimal digits as their ordinary span. Therefore, the span is dependent on the nature of the item, with the extra items beyond three being only capable of being stored if they have been learned in units associated with other items (Broadbent, 1971). One can argue that even if the amount of information that can be stored in STM is limited to three, this still represents a limited capacity.

According to the text comprehension mental model presented earlier, the propositional textbase is the middle layer in mental representation. As the propositions are processed by the comprehender, s/he must relate new propositions to previous ones (Kintsch and van Dijk, 1978). However, they estimate a memory capacity limit on the number of propositions that can be held of four. One consequence is that comprehension may be disrupted if the comprehender fails to relate a new proposition to earlier text because previous propositions are no longer active in memory. Recall that the propositional textbase has a parallel in procedural and object-oriented programming mental models, namely the program model (see Section 8.1). This suggests that memory capacity limit will have a direct consequence on the ability to comprehend object-oriented programs in the sense that it will constrain the construction of the program model.

8.3.3 Evidence Against STM

Studies in cognitive psychology provide evidence against the theory of STM. We start with the evidence that is frequently cited in favor of the theory of STM and then present disconfirmatory evidence.

The existence of STM is exemplified through memory span studies. In one such study (Shepard and Teghtsoonian, 1961) the subjects were given 200 three-digit numbers and were asked to identify when numbers were repeated. The experimenters were interested in the subjects' ability to recognize a repeated number as a function of how many other numbers intervened (i.e., the *lag*). It was demonstrated that as the lag increased the subjects' ability to correctly identify a repeated number decreased. This suggests that subjects tended to keep only the most recent numbers in STM since correct responses for the smallest lag tended to be highest, and that memory would get progressively worst as the numbers were pushed out of STM. Further evidence for STM came from studies of free recall in which subjects are presented with a list of unrelated words and asked to recall as many as possible in any order they wish. These showed that when recall was immediate, there is a tendency for the last few items to be very well recalled – the *recency effect*. After a brief delay the recency effect disappears (Baddeley, 1990). More evidence for separate STM and LTM came from brain-damaged patients and amnesics, where they would exhibit poor performance on STM tasks (such as memory span tests) but have normal long-term learning, or vice versa with impaired performance on delayed recall but good recency effects (Baddeley, 1990).

However, other experiments show that a similar forgetting function occurs for studied items over a period of days, weeks, or even decades (Anderson, 1995; Baddeley and Hitch, 1977). Therefore, there is nothing special about the STM recall function witnessed in recognition and recall experiments as the above. Furthermore, there have been studies that explicitly contradict the assumption of limited capacity in STM.

One would expect that if STM is filled to capacity, there would be very little possibility for doing other reasoning, retrieval, learning, and comprehending tasks at the same time. Baddeley and Hitch (Baddeley and Hitch, 1977) performed a study where subjects concurrently performed a

---

[16] Although it has been demonstrated that with appropriate chunking and retreival strategies, the digit span can be increased quite remarcably (Ericsson and Chase, 1982).

digit span task while being tested for free recall of unrelated words. They found that concurrent digit span impaired the long-term memory of learning but had no effect on recency. This is counter to the hypothesis that STM has limited capacity where new information pushes out old information. Baddeley (Baddeley, 1986) presents results from concurrent digit span and reasoning tasks. Up to eight digits were used which should overwhelm STM. The delay in performing the reasoning tasks was only 35% with 8 digits and the error rate remains relatively constant at 5%. Having to repeat a sequence of 8 digits should cause reasoning performance to break down completely, but it does not.

8.3.4 Working Memory

Instead of the traditional view of STM, Baddeley (Baddeley, 1986) has proposed a model of working memory in which a controlling central executive supervises and coordinates a number of subsidiary slave systems. Two slave systems examined were: (a) the *phonological loop*, which is assumed to be responsible for the manipulation of speech-based information, and (b) the *visuo-spatial sketchpad*, which is assumed to be responsible for setting up and manipulating visual images. The phonological loop is assumed to have two parts: a *phonological store* that is capable of holding speech-based information, and an *articulatory control process* based on inner speech. We will focus on the phonological loop since it is of most relevance to the cognitive complexity theories of object-oriented programs.

Baddeley et al. (Baddeley et al., 1975) and Vallar and Baddeley (Vallar and Baddeley, 1982) found that the amount of correct recall of a series of words depends on the number of syllables in the words. Thus, the memory span is controlled by the speed at which information can be rehearsed (Baddeley, 1986). Memory traces within the phonological store are assumed to fade and become unretrievable after 1.5 to 2 seconds. The memory trace can be refreshed by reading off the trace into the articulatory control process, which then feeds it back to the store, the process underlying subvocal rehearsal. Memory span is then determined by the number of items that can be refreshed before they fade away, which depends on how rapidly the trace fades and how long it takes to articulate each item and hence refresh the memory trace. The articulatory control process is also capable of taking written material, converting it into a phonological code and registering it in the phonological store (Baddeley, 1990).

The difference between the phonological loop and the common view of STM is that information does not have to spend time in the phonological loop to get into LTM. Rather, the phonological loop is just an auxiliary mechanism for keeping information available (Anderson, 1995).

8.3.5 Memory Span Revisited

The question remains as to the nature of the relationship between the phonological loop and memory span. Zhang and Simon (Zhang and Simon, 1985) performed a series of studies that address this issue. They used Chinese characters to measure memory span, and found that their overall results apply well to English materials. A simple hypothesis would state that memory span reflects a constant number of chunks. They found that is not so. For instance, in one study their subjects were presented with familiar Chinese radicals, words and characters each of which can be considered a single chunk. Radicals do not have commonly used oral names. Characters have single syllable pronunciations. Words have two syllable pronunciations. They found that span was largest for the characters, suggesting that span is a function of how long the material takes to articulate. The extent of familiarity affects the time it takes to articulate. Therefore, material that is highly familiar will result in a larger span. They propose the following span equation measured in number of chunks (C):

$$C = \frac{T}{a + b(S-1)} \qquad \textbf{Eqn. 22}$$

where T is a fixed time interval that reflects the duration of the underlying storage parameter, say 2 seconds, a is the amount of time to bring each new chunk into the articulatory mechanism,

b the time to articulate each syllable in the chunk beyond the first, and S is a measure of chunk complexity which is taken to be the average size of a chunk in syllables.

The memory constants used in the above equation reflect the recall of simple language constructs. It is questionnable whether the same values can be used when the chunk is an object-oriented class. Furthermore, other evidence (Just and Carpenter, 1992) suggests that memory capacity differs amongst individuals. If this is the case, then there is no single span that is applicable across the board, but rather each individual developer will have a different span.

The implication of this is that there is not a strong theoretical reason to believe that a fixed threshold for object-oriented metrics would be found. An object-oriented threshold means that the relationship between a metric and fault-proneness changes at the threshold value from being weak to being stronger. The issue of object-oriented thresholds has been discussed and evaluated recently in (Benlarbi et al., 2000; El-Emam et al., 2000a), and as expected, no evidence supporting thresholds was found. We therefore do not consider thresholds any further in this paper, and assume that there are no breakpoints in the relationship between object-oriented metrics and fault-proneness.

8.3.6 Activation Theory and LTM

As we saw above, rehearsal serves to keep information in working memory for as long as it is rehearsed, with the span being affected by speed of rehearsal. At some point, however, information must also be retained for a longer period of time.

Anderson (Anderson, 1993) has proposed activation theory to explain the workings of LTM. This stipulates that the strength of the trace to information in memory determines in part how active it can get and hence how accessible it will be. The amount of activation spread to a memory depends on the strength of that memory. Each time a memory trace is activated, it will increase a little in its strength. Strength of a trace can also be increased by repeated practice. Loss of information from memory is affected by the passage of time. Therefore, memory activation is determined by:

- How recently it was used, and
- How much of the memory was practiced (i.e., how frequently the same information was encountered)

Another factor, *interference effects* (Anderson, 1995), influences the ability to retain information. If intervening information is learned after some initial information, then, the retention of the initial information is not as strong.

The additional concept of *spreading activation* stipulates that the concepts in memory are connected in a network. Whenever a concept is activated, surrounding concepts are activated as well. For instance, if two concepts are associated, and if one of these concepts is activated, then the second concept is *primed*, meaning that the memory trace to the second concept has also been partially activated. In fact, studies show that in general the more associations to a piece of information that are learned, the more difficult it becomes to recall the information (Anderson, 1974). This is explained by the limited capacity of spreading activation. Therefore, the more paths from a concept, the less activation will go down any path. This is known as the *fan effect*.

8.4 Relationship Between Dependencies and Recall

8.4.1 Familiarity

It was noted above that the more a memory is practiced (i.e., it becomes more familiar) the stronger the trace to that memory. This mechanism can be formulated in terms of object-oriented applications.

One factor that is contended to have an impact on program complexity is chunk familiarity (Henderson-Sellers, 1996). It is argued that chunks that are referenced more often will be more familiar since they are used more often. Davis also argues that frequency of reference is related

with familiarity for procedural programs (Davis, 1984). Therefore, metrics that are potential indicators of familiarity would be predicted to be positively associated with the comprehension of the classes.

Burkhardt et al. (Burkhardt et al., 1998) performed a detailed study of object-oriented program comprehension. Many of the results from this study are relevant for the following discussion, and therefore they are described below. They recorded the files accessed while subjects were studying a C++ program, as well as verbal protocols. They also asked their subjects to subsequently perform a reuse and documentation task. The subjects consisted of both novices and professionals, but we only focus on the professional results. The program had an inheritance hierarchy with one root, as well as a number of other classes not within the inheritance hierarchy.

They found that the most consulted classes were at the top of the inheritance hierarchy, at the bottom of the inheritance hierarchy (i.e., deepest classes), and those that provide services used by most other classes but were not in the main inheritance hierarchy. The classes lowest in the hierarchy are the ones first referenced in the "main" function. This indicates that the classes with zero values of DIT (i.e., root classes), with the highest values on DIT, and those with high values of export coupling are consulted more frequently, and therefore one can further argue that they would be quite familiar to software engineers. The classes that were least consulted tended to be in the middle of the inheritance hierarchy.

Another interesting observation was made when the session was divided into three chronological stages. At stages 2 and 3 classes up the inheritance hierarchy whose methods are used frequently by its children are consulted often. This suggests that higher values of NOC mean higher familiarity since the more children the more likely that a parent will be consulted frequently. Furthermore, this also suggests that classes with high values of descendant-based export coupling will be consulted often and hence be more familiar.

We can conclude from the above that the following are characteristics of classes that would be most familiar to engineers since they are consulted often:

- Classes at the root of the inheritance hierarchy (DIT=0)
- Classes at the bottom of the inheritance hierarchy (largest DIT)
- Classes with many children (large NOC)
- Classes with high values of "Other" and "Descendant" based export coupling

8.4.2 Tracing

We noted above that interference due to having to learn intervening material and fan-effects reduce recall ability.

Henderson-Sellers (Henderson-Sellers, 1996) notes that tracing disrupts the process of chunking. This occurs when it becomes necessary to understand another chunk, as when a method calls another method in a different class, or when an inherited property needs to be understood. Such disruptions cause interference, and increases the chances that knowledge of the original chunk is lost. In fact, tracing dependencies is a common task when understanding object-oriented software. Wilde et al. (Wilde et al., 1993) found that programmers have to understand a method's context of use by tracing back through the chain of calls that reach it, and tracing the chain of methods it uses. Their findings were from an interview study of two C++ object-oriented systems at Belicore and a PC Smalltalk environment. The three systems investigated span different application domains. Furthermore, they found that one has to trace inheritance dependencies, which may be considerably complicated due to dynamic binding. A similar point was made in (Leijter et al., 1992) about the understandability of programs in such languages as C++ that support dynamic binding.

In one study of how programmers comprehend code (Koenemann and Robertson, 1991), it was found that they followed an "as-needed" strategy. This means that subjects restrict their understanding to parts of the code they consider to be relevant to the task at hand. Their task

consisted of making modifications. The authors divided the code into that which is *directly relevant* to the modification task. Directly relevant code was examined the most. Code of *intermediate relevance* was examined to a lesser extent. Subjects studied code of intermediate relevance if they discover an interaction that they judge needs to be understood to complete the task. It was noted that they required tracing procedure and function calls backwards and forwards. Although this study was on a procedural program, the same behavior as the above two object-oriented studies was exhibited. Furthermore, in an object-oriented study, a similar behavior was observed (Corritore and Wiedenbeck, 2000).

At a global level, Burkhardt et al. (Burkhardt et al., 1998) found that experts tended to follow a number of different strategies for tracing through a program:

Random: A random strategy is defined as one where the subject does not read according to information relationships in the program (e.g., inheritance and composition), but rather in a manner that is arbitrary with respect to the relationships. For example, in the order of the on-line listing of files. The use of this strategy is strong initially, but decreases over time, suggesting that the experts were orienting themselves to the overall application at the start.

Methods: This strategy involves trying to understand the internal workings of a method by systematically going through its code. While the use of this strategy was weak initially, it dominated towards the end of the session.

Execution: With the execution strategy the subject follows the order of execution of the program, i.e., a mental simulation of the program. Therefore call sequences to other methods are followed systematically. Initially, this strategy is used often, but its usage diminishes over time.

Inheritance: With the inheritance strategy the subjects follow the inheritance hierarchy either up or down. Initially the inheritance strategy is not used often, but its usage increases in later stages of comprehension.

Composition: Composition relationships occur when a class contains attributes whose types are other classes. An approach of reading guided by composition relationships would be indicated by a shift of attention from one class to another when such an attribute is encountered. Tracing through composition is used infrequently through all comprehension stages, but is used nevertheless.

Variables: With this strategy the subjects attempt to find information about a variable or to follow the use of a variable, i.e., it involves tracing the flow of data through a class. It may also cross class boundaries if a variable is passed as an argument to a method of another class. This strategy is used infrequently at the outset, but its usage increases gradually.

At the design phase of a project inheritance and composition information is readily available. Other information required to follow the methods, execution, and variables strategies will frequently not be available until detailed design and implementation. Therefore, for the purpose of design metrics we focus on the former two. The authors noted that overall subjects used "static" strategies, namely inheritance and composition, more often than dynamic strategies, such as execution (43% vs. 29%).

As we saw earlier, subjects did trace up the inheritance hierarchy when parent services were used frequently. This suggests that tracing will be more frequent from classes with high "Ancestor" based import coupling. Furthermore, tracing composition relationships suggests that import coupling to "Other" classes leads to following a trace to other classes. Therefore, one can conclude that more interference will occur with classes with high values for these types of import coupling.

Furthermore, it can be argued that the fan effect will be exacerbated when a class has a high value of import coupling. This is because the class will have many other associated classes whose services have to be understood in order to understand what the class is doing.

8.5 Derivation of Hypotheses

8.5.1 Export Coupling Hypotheses

The cognitive theory predicts that classes with high export coupling will be more familiar, and hence less fault-prone. One can also argue for another, non-cognitive, effect in the same direction. Classes with large export coupling can be seen as central or core to a particular application. The designers of a system will know that, and therefore they may deliberately give it more attention. For example, more experienced programmers may be asked to work on these, more test cases are developed for classes with frequently used services, they may receive more thorough inspections (say larger inspection teams) and programmers are more careful during implementation. This will lead to fewer faults for high export coupling classes. One survey of 40 programmers found that the majority regarded it critical to test the most used components (Duncan and Robson, 1996). This response was strongest for the more experienced programmers. Although this study was performed with C programmers, it indicates that components with high export coupling are likely to receive more attention.

In fact, the evidence to date demonstrates exactly the opposite effect: a strong positive relationship between export coupling and (post-release) fault-proneness (El-Emam et al., 1999; El-Emam et al., 2001b). A possible explanation is that a client of a class d makes assumptions about d's behavior. A class with more export coupling has more clients and therefore more assumptions are made about its behavior due to the existence of more clients. Since the union of these assumptions can be quite large, it is more likely that this class d will have a subtle fault that violates this large assumption space, compared to other classes with a smaller set of assumptions made about their behavior. This is similar to the argument made in (Briand et al., 2000) for export coupling metrics.[17] This can be interpreted to be a fan effect. Another possible explanation is an interference effect. It was noted earlier from the study of Wilde et al. (Wilde et al., 1993) that engineers trace back the users of a class during comprehension. This, therefore suggests that classes with higher export coupling will suffer interference and fan effects.

The above exposition suggests a number of competing hypotheses about the impact of export coupling on fault-proneness. The competing hypotheses for the export coupling metrics are:

- **H-EC+ : Positive Association** Classes with high export coupling have many assumptions made about their behavior. This makes it easier to violate some of these assumptions. This can be seen as a fan effect. Furthermore, more interference will occur as the engineers trace back frequently from classes with high export coupling to understand how they are used. Given that these mechanisms are in the same direction, it is not possible to disentangle them. We therefore consider them together as one hypothesis.
- **H-EC- : Negative Association** Classes with high export coupling are more familiar because they are consulted often. Furthemore, classes with high export coupling are given extra attention during development. Given that these two mechanisms are in the same direction, it is not possible to disentangle them. Therefore, by definition it is not possible in an observational study to determine which one of these two mechanisms is operating. We therefore consider them together as one hypothesis.

8.5.2 Inheritance – Number of Children Hypotheses

The above theoretical exposition indicates that classes with high values of NOC would be very familiar and therefore easier to understand. The consequence is that fewer faults will be injected and hence fewer post-release faults will be detected. One would therefore expect a negative relationship between NOC and post-release faults. However, the premise of NOC's effect is that classes with high NOC will also have high values on Descendant-based export coupling, and that higher descendant-based export coupling means more familiarity and therefore a smaller fault-

[17] It is arguable that if a fault is due to class d violating an assumption made about it, then this is a problem in d's client rather than d itself since the client is not using d properly. However, in practice a fault is logged against the class that is changed to fix the fault. If d is changed then this is a strong indicator that there was a problem in d.

proneness. This is a clear confounding effect. If we remove the effect of export coupling from the mechanism, then there is no cognitive theory to explain why NOC should be associated with fault-proneness. Two studies that empirically evaluated NOC found no relationship between NOC and post-release faults (El-Emam et al., 1999; El-Emam et al., 2001b), hence supporting our contention. Therefore, we hypothesise no relationship between NOC and fault-proneness.

8.5.3 Inheritance – Depth of Inheritance Tree Hypotheses

As noted in (Deligiannis and Shepperd, 1999), the software engineering community has been preoccupied with inheritance and its effect on quality. Inheritance is strongly believed to make the understandability of object-oriented software difficult. According to a survey of object-oriented practitioners (Daly et al., 1995a), 55% of respondents agree that inheritance depth is a factor in understanding object-oriented programs (Daly et al., 1995a). The cognitive theory above suggests that classes at the root and deepest in the inheritance hierarchy are likely to be familiar since they are consulted often.

A number of empirical studies have investigated this issue, although some of them do not test this hypothesis directly. An experimental investigation found that making changes to a C++ program with inheritance consumed more effort than a program without inheritance, and the author attributed this to the subjects finding the inheritance program more difficult to understand based on responses to a questionnaire (Cartwright, 1998). In two further experiments (Unger and Prechelt, 1998), subjects were given three equivalent Java programs to modify, and the maintenance time was measured. One of the Java programs was *flat*, in that it did not take advantage of inheritance; one had an inheritance depth of 3; and one had an inheritance depth of 5. In an initial experiment, the programs with an inheritance depth of 3 on the average took longer to maintain than the flat program, but the program with an inheritance depth of 5 took as much time as the flat program. The authors attribute this to the fact that the amount of changes required to complete the maintenance task for the deepest inheritance program was smaller. The results for a second task in the first experiment and the results of the second experiment indicate that it took longer to maintain the programs with inheritance.

However, further studies cast doubt on the above conclusions. Daly et al. (Daly et al., 1996) contradict these findings. The authors conducted a series of classroom experiments comparing the time to perform maintenance tasks on a 'flat' C++ program and a C++ program with three levels of inheritance. The result was a significant reduction in maintenance effort for the inheritance program. An internal replication by the same authors found the results to be in the same direction, albeit the p-value was larger. This suggests an inverse effect for inheritance depth to the one described above.

However, the above studies have the unit of analysis the program rather than the class. Therefore, although they show that no-inheritance results in better performance, they do not show that classes at the root of a hierarchy are indeed easier to understand.

A non-experimental study by Cartwright and Shepperd (Cartwright and Shepperd, 2000) found that classes with inheritance tend to be more fault prone. Another study (El-Emam et al., 2001b) found that there is a relationship between the depth of inheritance tree and fault-proneness in Java programs. Therefore, classes at the root of the tree had the least fault-proneness. This evidence suggests that root classes are more familiar and hence easier to understand, which is consistent with predictions from the cognitive theory.

Two recent studies reported no relationship between inheritance depth and fault-proneness in C++ programs (El-Emam et al., 1999, 2000b). These two studies were performed with commercial applications. One reason for not finding a relationship in the non-experimental studies is that it is not uncommon for inheritance hierarchies to be shallow, with inheritance not being used extensively (Cartwright and Shepperd, 2000; Chidamber and Kemerer, 1994; Chidamber et al., 1998; El-Emam et al., 1999, 2000b). If there is little variation in the DIT metric then it is unlikely that a strong relationship with fault-proneness will be found.

The above review makes clear that the prediction of whether root classes are indeed less fault-prone has not been adequately tested, and therefore requires additional empirical study.

Furthermore, for the second prediction from the cognitive theory, that classes deepest in the hierarchy will be easier to understand, there has been equivocal evidence. There is also a competing explanation as to why deeper classes may in fact not be easier to understand. In a set of interviews with 13 experienced users of object-oriented programming, Daly et al. (Daly et al., 1995b) noted that if the inheritance hierarchy is designed properly then the effect of distributing functionality over the inheritance hierarchy would not be detrimental to understanding. However, it has been argued that there exists increasing conceptual inconsistency as one travels down an inheritance hierarchy (i.e., deeper levels in the hierarchy are characterized by inconsistent extensions or specializations of super-classes) (Dvorak, 1994). Therefore inheritance hierarchies are likely to be improperly designed in practice. One study by Dvorak' (Dvorak, 1994) supports this argument. Dvorak found that subjects were more inconsistent in placing classes deeper in the inheritance hierarchy than they were in placing them at higher levels in the inheritance hierarchy. Such inconsistency may make the deepest classes less understandable and hence more fault-prone.

The competing hypotheses for the DIT metric are therefore:

- **H-DITQ: Qudratic Association** Classes at the root are consulted more often, and therefore they are more familiar. Classes deeper in the hierarchy are also consulted more often. Classes in the middle of the inheritance hierarchy are, in general, not consulted often and therefore they are not familiar. This suggests a quadratic relation between DIT and fault-proneness.
- **H-DIT+: Linear Positive Association** Root classes are most familiar, but the hierarchy is not well constructed, and therefore there exists higher conceptual entropy at the deepest classes. This will result in the root classes having few faults and the deeper classes having the most faults.

8.5.4 Import Coupling Hypotheses

The theory above predicts a positive relationship between import coupling and fault-proneness due to interference and fan effects. The evidence of this, however, is equivocal in that some studies demonstrate a positive association and others no association (El-Emam et al., 1999; El-Emam et al., 2001b). This includes both ancestor-based import coupling and import coupling to other classes.

For import coupling, we have only one hypothesis:

H-IC+: Positive Association Due to interference and fan effects, there will be a positive association between import coupling and fault-proneness.

9 Appendix C: Evaluating Prediction Accuracy

It is common that prediction models using object-oriented metrics are cast as a binary classification problem. We first present some notation before discussing the binary accuracy measure that we use.

Table 16 shows the notation in obtained frequencies when a binary classifier is used to predict the class of unseen observations in a confusion matrix. We consider a class as being high risk if it has a fault and low risk if it does not have a fault.

		Predicted Risk: Low	Predicted Risk: High	
Real Risk	**Low**	n_{11}	n_{12}	N_{1*}
	High	n_{21}	n_{22}	N_{2*}
		N_{*1}	N_{*2}	N

Table 16: Notation for a confusion matrix. This is similar to Table 4 except that we use a more general terminology.

Such a confusion matrix also appears frequently in the medical sciences in the context of evaluating diagnostic tests, for example, see (Gordis, 1996). Two important parameters have been defined on such a matrix that will be used for our exposition, namely sensitivity and specificity.

The *sensitivity* of a binary classifier is defined as:

$$s = \frac{n_{22}}{n_{21} + n_{22}} \qquad \textbf{Eqn. 23}$$

This is the proportion of high risk classes that have been correctly classified as high risk classes.

The *specificity* of a binary classifier is defined as:

$$f = \frac{n_{11}}{n_{11} + n_{12}} \qquad \textbf{Eqn. 24}$$

This is the proportion of low risk classes that have been correctly classified as low risk classes.

Ideally, both the sensitivity and specificity should be high. A low specificity means that there are many low risk classes that are classified as high risk. Therefore, the organization would be wasting resources reinspecting or focusing additional testing effort on these classes. A low sensitivity means that there are many high risk classes that are classified as low risk. Therefore, the organization would be passing high risk classes to subsequent phases or delivering them to the customer. In both cases the consequences may be expensive field failures or costly defect correction later in the life cycle.

The J coefficient of Youdon (Youden, 1950) was suggested in (El-Emam et al., 2001a) as an appropriate measure of accuracy for binary classifiers in software engineering. This is defined as:

$$J = s + f - 1 \qquad \textbf{Eqn. 25}$$

This coefficient has a number of desirable properties. First, it is prevalence independent (i.e., it does not depend on the proportion of faulty classes in the data set). For example, if our classifier has specificity and sensitivity equal to $f = 0.9$ and $s = 0.7$, then its J value is 0.6 irrespective of prevalence. The J coefficient can vary from minus one to plus one, with plus one being perfect accuracy and –1 being the worst accuracy. A guessing classifier (i.e., one that guesses High/Low risk with a probability of 0.5) would have a J value of 0. Therefore, J values greater than zero indicate that the classifier is performing better than would be expected from a guessing classifier.

However, in order to use the J coefficient it is necessary to decide on a cutoff value from the logistic regression model. Recall that a logistic regression model makes predictions as a probability rather than a binary value (i.e., if we use a LR model to make a prediction, the predicted value is the probability of the occurrence of a fault). It is common to choose a cutoff value for this predicted probability. For instance, if the predicted probability is greater than 0.5 then the class is predicted to be high risk.

Previous studies have used a plethora of logistic regression cutoff values to decide what is high risk or low risk, for example, 0.5 (Basili et al., 1996; Briand et al., 1998b, 1999b; Morasca and Ruhe, 1997), 0.6 (Briand et al., 1998b), 0.65 (Briand et al., 1998b; Briand et al., 2000), 0.66 (Briand et al., 1998a), 0.7 (Briand et al., 2000), and 0.75 (Briand et al., 2000). In fact, and as noted by some authors (Morasca and Ruhe, 1997), the choice of cutoff value is arbitrary, and one can obtain different results by selecting different cutoff values, for example, see (El-Emam et al., 1999).

A general solution to the arbitrary thresholds problem mentioned above is Receiver Operating Characteristic (ROC) curves (Metz, 1978). One selects many cutoff points, from 0 to 1 in our case, and calculates the sensitivity and specificity for each cutoff value, and plots sensitivity against 1-specificity as shown in Figure 11. Such a curve describes the compromises that can be made between sensitivity and specificity as the cutoff value is changed. One advantage of expressing the accuracy of our prediction model (or for that matter any diagnostic test) as an ROC curve is that it is independent of the cutoff value, and therefore no arbitrary decisions need be made as to where to cut off the predicted probability to decide that a class is high risk (Zweig and Campbell, 1993). Furthermore, using an ROC curve, one can easily determine the optimal operating point, and hence obtain an optimal cutoff value for an LR model.

For our purposes, we can obtain a summary accuracy measure from an ROC curve by calculating the area under the curve using a trapezoidal rule (Hanley and McNeil, 1982). The area under the ROC curve has an intuitive interpretation (Hanley and McNeil, 1982; Spiegelhalter, 1986): it is the estimated probability that a randomly selected class with a fault will be assigned a higher predicted probability by the logistic regression model than another randomly selected class without a fault. Therefore, an area under the curve of say 0.8 means that a randomly selected faulty class has an estimated probability larger than a randomly selected not faulty class 80% of the time.

When a model cannot distinguish between faulty and not faulty classes, the area will be equal to 0.5 (the ROC curve will coincide with the diagonal). When there is a perfect separation of the values of the two groups, the area under the ROC curve equals 1 (the ROC curve will reach the upper left corner of the plot).

Therefore, to compute the accuracy of a prediction logistic regression model, we use the area under the ROC curve, which provides a general and non-arbitrary measure of how well the probability predictions can rank the classes in terms of their fault-proneness.

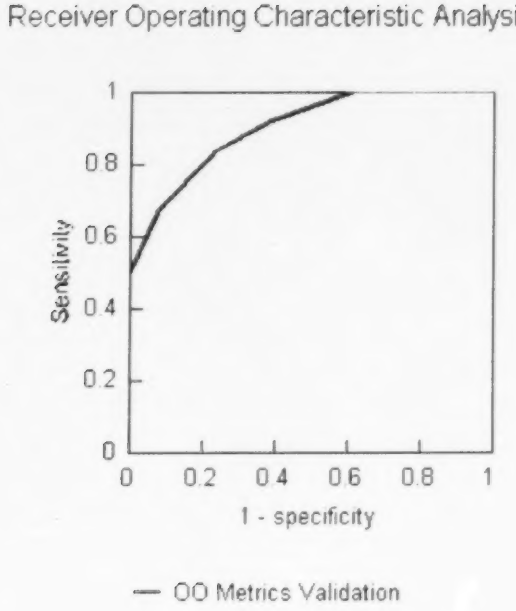


Figure 11: Hypothetical example of an ROC curve.

10 References

Abreu, F. Brite e, Carapuca, R. (1994). Object-Oriented Software Engineering: Measuring and Controlling the Development Process. Proceedings of the 4th International Conference on Software Quality.

Anderson, J., (1974). Retrieval of Propositional Information from Long Term Memory. Cognitive Psychology, 6, 451-474.

Anderson, J. (1993). Rules of the Mind. Erlbaum.

Anderson, J. (1995). Cognitive Psychology and Its Implications. W. H. Freeman and Company.

Baddeley, A., Thompson, N., Buchanan, M., (1975). Word Length and the Structure of Short-Term Memory. Journal of Verbal Learning and Verbal Behavior, 14, 575-589.

Baddeley, A., Hitch, G. (1977). Recency Re-examined. In: S. Dornic (eds.), Attention and Performance VI, Lawrence Erlbaum Associates.

Baddeley, A. (1986). Working Memory. Oxford University Press.

Baddeley, A. (1990). Human Memory: Theory and Practice. Lawrence Erlbaum Associates.

Basili, V., Briand, L., Melo, W., (1996). A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering, 22:10, 751-761.

Belsley, D., Kuh, E., Welsch, R. (1980). Regression Diagnostics: Identifying Influential Data and Sources of Collinearity. John Wiley and Sons.

Benlarbi, S., Melo, W. (1999). Polymorphism Measures for Early Risk Prediction. Proceedings of the 21st International Conference on Software Engineering, 334-344.

Benlarbi, S., El-Emam, K., Goel, N., Rai, S. (2000). Thresholds for Object-Oriented Measures. Proceedings of the International Symposium on Software Reliability Engineering (to appear).

Bergantz, D., Hassell, J., (1991). Information Relationships in PROLOG Programs: How do Programmers Comprehend Functionality ? International Journal of Man-Machine Studies, 35, 313-328.

Binkley, A., Schach, S. (1998). Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures. Proceedings of the 20th International Conference on Software Engineering, 452-455.

Boehm-Davis, D., Holt, R., Schultz, A., (1992). The Role of Program Structure in Software Maintenance. International Journal of Man-Machine Studies, 36, 21-63.

Bransford, J., Barclay, J., Franks, J., (1972). Sentence Memory: A Constructive Versus Interpretive Approach. Cognitive Psychology, 3, 193-209.

Briand, L., Devanbu, P., Melo, W. (1997). An Investigation into Coupling Measures for C++. Proceedings of the 19th International Conference on Software Engineering.

Briand, L., Daly, J., Porter, V., Wuest, J. (1998a). Predicting Fault-Prone Classes with Design Measures in Object Oriented Systems. Proceedings of the International Symposium on Software Reliability Engineering, 334-343.

Briand, L., Wuest, J., Ikonomovski, S., Lounis, H., (1998b). A Comprehensive Investigation of Quality Factors in Object-Oriented Designs: An Industrial Case Study. International Software Engineering Research Network, ISERN-98-29.

Briand, L., Daly, J., Wuest, J., (1999a). A Unified Framework for Coupling Measurement in Object-Oriented Systems. IEEE Transactions on Software Engineering, 25:1, 91-121.

Briand, L., Wuest, J., Ikonomovski, S., Lounis, H. (1999b). Investigating Quality Factors in Object-Oriented Designs: An Industrial Case Study. Proceedings of the International Conference on Software Engineering.

Briand, L., Wuest, J., Daly, J., Porter, V., (2000). Exploring the Relationships Between Design Measures and Software Quality in Object Oriented Systems. Journal of Systems and Software, 51, 245-273.

Broadbent, D. (1971). Decision and Stress. Academic Press.

Broadbent, D. (1975). The Magic Number Seven After Fifteen Years. In: A. Kennedy (eds.), Studies in Long Term Memory, Wiley.

Brooks, R., (1983). Towards a Theory of the Comprehension of Computer Programs. International Journal of Man-Machine Studies, 18, 543-554.

Brooks, R., (1999). Towards a Theory of the Cognitive Pocesses in Computer Programming. International Journal of Human-Computer Studies, 51, 197-211.

Burkhardt, J-M, Detienne, F., Wiedenbeck, S. (1997). Mental Representations Constructed by Experts and Novices in Object-Oriented Program Comprehension. Human-computer Interaction: INTERACT'97, 339-346.

Burkhardt, J-M, Detienne, F., Wiedenbeck, S. (1998). The Effect of Object-Oriented Programming Expertise in Several Dimensions of Comprehension Strategies. Proceedings of the 6th International Workshop on Program Comprehension, 82-89.

Burkhardt, J-M., Detienne, F., Wiedenbeck, S., (1999). Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase. Submitted for Publication.

Cant, S., Henderson-Sellers, B., Jeffery, R., (1994). Application of Cognitive Complexity Metrics to Object-Oriented Programs. Journal of Object-Oriented Programming, 7:4, 52-63.

Cant, S., Jeffery, R., Henderson-Sellers, B., (1995). A Conceptual Model of Cognitive Complexity of Elements of the Programming Process. Information and Software Technology, 7, 351-362.

Cartwright, M., (1998). An Empirical View of Inheritance. Information and Software Technology, 40, 795-799.

Cartwright, M., Shepperd, M., (2000). An Empirical Investigation of an Object-Oriented Software System. IEEE Transactions on Software Engineering (to appear).

Chidamber, S., Kemerer, C., (1994). A Metrics Suite for Object-Oriented Design. IEEE Transactions on Software Engineering, 20:6, 476-493.

Chidamber, S., Kemerer, C., (1995). Authors' Reply. IEEE Transactions on Software Engineering, 21:3, 265.

Chidamber, S., Darcy, D., Kemerer, C., (1998). Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis. IEEE Transactions on Software Engineering, 24:8, 629-639.

Corritore, C., Wiedenbeck, S., (1999). Mental Representations of Expert procedural and Object-Oriented Programmers in a Software maintenance Task. International Journal of Human-Computer Studies, 50, 61-83.

Corritore, C., Wiedenbeck, S. (2000). Direction and Scope of Comprehension-Related Activities by Procedural and Object-Oriented Programmers: An Empirical Study. Proceedings of the 8th International Workshop on Program Comprehension, 139-148.

Curtis, B. (1979). In Search of Software Complexity. Proceedings of the IEEE Workshop on Quantitative Software Models, 95-106.

Daly, J., Miller, J., Brooks, A., Roper, M., Wood, M., (1995a). Issues on the Object-Oriented Paradigm: A Questionnaire Survey. Department of Computer Science - University of Strathclyde, EFoCS-8-95.

Daly, J., Wood, M., Brooks, A., Miller, J., Roper, M., (1995b). Structured Interviews on the Object-Oriented Paradigm. Department of Computer Science - University of Strathclyde, EFoCS-7-95.

Daly, J., Brooks, A., Miller, J., Roper, M., Wood, M., (1996). Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software. Empirical Software Engineering: An International Journal, 1:2, 109-132.

Daneman, M., Carpenter, P., (1980). Individual Differences in Working Memory and Reading. Journal of Verbal Learning and Verbal Behavior, 19, 450-466.

Davis, J., (1984). Chunks: A Basis for Complexity Measurement. Information Processing & Management, 20:1, 119-127.

Deligiannis, I., Shepperd, M. (1999). A Review of Experimental Investigations into Object-Oriented Technology. Proceedings of the Fifth IEEE Workshop on Empirical Studies of Software Maintenance, 6-10.

Derksen, S., Keselman, H., (1992). Backward, Forward and Stepwise Automated Subset Selection Algorithms: Frequency of Obtaining Authentic and Noise Variables. British Journal of Mathematical and Statistical Psychology, 45, 265-282.

Dooling, D., Christiaansen, R., (1977). Episodic and Semantic Aspects of Memory for Prose. Journal of Experimental Psychology: Human Learning and Memory, 3, 428-436.

Duncan, I., Robson, D., (1996). An Exploratory Study of Common Coding Faults in C Programs. Software Maintenance: Research and Practice, 8, 241-256.

Dunsmore, A., Roper, M., Wood, M. (2000). Object-Oriented Inspection in the Face of Delocalization. Proceedings of the International Conference on Software Engineering, 467-476.

Dvorak, J., (1994). Conceptual Entropy and Its Effect on Class Hierarchies. IEEE Computer, 59-63.

Ehrlich, K., Soloway, E. (1984). An Empirical Investigation of the Tacit Plan Knowledge in Programming. In: J. Thomas and M. Schneider (eds.), Human Factors in Computer Systems.

El-Emam, K., Benlarbi, S., Goel, N., Rai, S., (1999). A Validation of Object-Oriented Metrics. National Research Council of Canada, NRC/ERB 1063.

El-Emam, K., Benlarbi, S., Goel, N., Melo, W., Lounis, H., Rai, S., (2000a). The Optimal Class Size for Object-Oriented Software: A Replicated Study. National Research Council of Canada, NRC/ERB 1074.

El-Emam, K., Benlarbi, S., Goel, N., Rai, S., (2000b). The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. IEEE Transactions on Software Engineering (to appear).

El-Emam, K., Benlarbi, S., Goel, N., Rai, S., (2001a). Comparing Case-Based Reasoning Classifiers for Predicting High Risk Software Components. Journal of Systems and Software (to appear).

El-Emam, K., Melo, W., Machado, J., (2001b). The Prediction of Faulty Classes Using Object-Oriented Design Metrics. Journal of Systems and Software (to appear).

Ericsson, K., Chase, W., (1982). Exceptional Memory. American Scientist, 70, 607-615.

Farnese, R., Melo, W., Veiga, A. da, (1999). JMetrics: Java Metrics Extractor. Oracle Consulting Services Brazil.

Fenton, N. (1991). Software Metrics: A Rigorous Approach. Chapman & Hall.

Fenton, N., Neil, M., (1999). Software Metrics: Successes, Failures, and New Directions. Journal of Systems and Software, 47, 149-157.

Fenton, N., Ohlsson, N., (2000). Quantitative Analysis of Faults and Failures in a Complex Software System. IEEE Transactions on Software Engineering (to appear).

Flack, V., Chang, P., (1986). Frequency of Selecting Noise Variables in Subset Regression Analysis: A Simulation Study.

Franz, L., Shih, J., (December 1994). Estimating the Value of Inspections and Early Testing for Software Projects. Hewlett-Packard Journal, 60-67.

Furuyama, T., Arai, Y., Iio, K., (1994). Fault Generation Model and Mental Stress Effect Analysis. Journal of Systems and Software, 26, 31-42.

Furuyama, T., Arai, Y., Iio, K., (1997). Analysis of Fault Generation Caused by Stress During Software Development. Journal of Systems and Software, 38, 13-25.

Gordis, L. (1996). Epidemiology. W. B. Saunders Company.

Hanley, J., McNeil, B., (1982). The Meaning and Use of the Area Under a Receiver Operating Characteristic Curve. Diagnostic Radiology, 143:1, 29-36.

Harrell, F., Lee, K., (1984). Regression Modelling Strategies for Improved Prognostic Prediction. Statistics in Medicine, 3, 143-152.

Harrell, F., Lee, K., Mark, D., (1996). Multivariate Prognostic Models: Issues in Developing Models, Evaluating Assumptions and Adequacy, and Measuring and Reducing Errors. Statistics in Medicine, 15, 361-387.

Hatton, L., (1997). Re-examining the Fault Density - Component Size Connection. IEEE Software, 89-97.

Hatton, L., (1998). Does OO Sync with How We Think ? IEEE Software, 46-54.

Henderson-Sellers, B. (1996). Object-Oriented Metrics: Measures of Complexity. Prentice-Hall.

Hilgard, E., Atkinson, R., Atkinson, R. (1971). Introduction to Psychology. Harcourt Brace Jovanovich.

Hosmer, D., Lemeshow, S. (1989). Applied Logistic Regression. John Wiley & Sons.

ISO/IEC-14598-1, (1996). Information Technology - Software Product Evaluation; Part 1: Overview. International Organization for Standardization and the International Electrotechnical Commission, ISO/IEC DIS 14598-1.

Johnson, P., Tjahjono, D., (1998). Does Every Inspection Really Need a Meeting ? Empirical Software Engineering: An International Journal, 3, 9-35.

Just, M., Carpenter, P., (1992). A Capacity Theory of Comprehension: Individual Differences in Working Memory. Psychological Review, 99:1, 122-149.

Khoshgoftaar, T., Allen, E., Jones, W., Hudepohl, J. (1998). Return on Investment of Software Quality Predictions. Proceedings of the IEEE Workshop on Application-Specific Software Engineering Technology, 145 -150.

Kintsch, W., van Dijk, T., (1978). Toward A Model of Text Comprehension and Reproduction. Psychological Review, 85, 363-394.

Kintsch, W., (1986). Learning from Text. Cognition and Instruction, 3, 87-108.

Koenemann, J., Robertson, S. (1991). Expert Problem Solving Strategies for Program Comprehension. Proceedings of the CHI'91 Conference, 125-130.

Kusumoto, S., (1993). Quantitative Evaluation of Software Reviews and Testing Processes. PhD. Thesis, Osaka University.

Laitenberger, O., Atkinson, C., Schlich, M., El-Emam, K., (2000). An Experimental Comparison of Reading Techniques for Defect Detection in UML Design Documents. Journal of Systems and Software, 53:2, 183-204.

Landwehr, J., Pergibon, D., Shoemaker, A., (1984). Graphical Methods for Assessing Logistic Regression Models. Journal of the American Statistical Association, 79:385, 61-71.

Leijter, M., Meyers, S., Reiss, S., (1992). Support for Maintaining Object-Oriented Programs. IEEE Transactions on Software Engineering, 18:12, 1045-1052.

Li, W., Henry, S., (1993). Object-Oriented Metrics that Predict Maintainability. Journal of Systems and Software, 23, 111-122.

Lorenz, M., Kidd, J. (1994). Object-Oriented Software Metrics. Prentice-Hall.

Metz, C., (1978). Basic Principles of ROC Analysis. Seminars in Nuclear Medicine, VIII:4, 283-298.

Miller, G., (1957). The Magical Number 7 Plus or Minus Two: Some Limits on Our Capacity for Processing Information. Psychological Review, 63, 81-97.

Morasca, S., Ruhe, G. (1997). Knowledge Discovery from Software Engineering Measurement Data: A Comparative Study of Two Analysis Techniques. Proceedings of the International Conference on Software Engineering and Knowledge Engineering.

Munson, J., Khoshgoftaar, T., (1992). The Detection of Fault-Prone Programs. IEEE Transactions on Software Engineering, 18:5, 423-433.

Nielsen, J., Richards, J., (1989). Experience of Learning and Using Smalltalk. IEEE Software, 73-77.

Pennington, N. (1987a). Comprehension Strategies in Programming. Empirical Studies of Programmers, 2nd Workshop, 100-113.

Pennington, N., (1987b). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. Cognitive Psychology, 19, 295-341.

Prechelt, L., Unger, B., Philippsen, M., Tichy, W., (1999). A Controlled Experiment on Inheritance Depth as a Cost Factor for Maintenance. Fakultaet fur Informatik, University of Karlsruhe, Germany.

Rist, R. (1996). System Structure and Design. Proceedings of the Workshop on Empirical Studies of Programmers, 163-194.

Schmalhofer, F., Glavanov, D., (1986). Three Components of Understanding A Programmer's Manual: Verbatim, Propositional, and Situational Represenations. Journal of Memory and Language, 25, 295-313.

Schneiderman, B., (1977). Measuring Computer Program Quality and Comprehension. International Journal of Man-Machine Studies, 9, 465-478.

Sebrechts, M., Black, J., (1982). Software Psychology: A Rich New Domain for Applied Psychology. Applied Psycholinguistics, 3, 223-232.

Shaft, T., Vessey, I., (1995). The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension. Information Systems Research, 6:3, 286-299.

Shepard, R., Teghtsoonian, M., (1961). Retention of Information Under Conditions Approaching a Steady State. Journal of Experimental Psychology, 62, 302-309.

Siy, H., Votta, L., (1999). Does the Modern Code Inspection Have Value ? Submitted for Publication.

Soloway, E., Ehrlich, K., (1984). Empirical Studies of Programming Knowledge. IEEE Transactions on Software Engineering, 10:5, 595-609.

Spiegelhalter, D., (1986). Probabilistic Prediction in Patient Management in Clinical Trials. Statistics in Medicine, 5, 421-433.

Spiro, R. (1977). Constructing a Theory of Reconstructive Memory: The State of the Schema Approach. In: R. Anderson, R. Spiro and W. Montague (eds.), Schooling and the Acquisition of Knowledge.

Sulin, R., Dooling, D., (1974). Intrusion of a Thematic Idea in Retention of Prose. Journal of Experimental Psychology, 103, 255-262.

Tang, M-H., Kao, M-H., Chen, M-H. (1999). An Empirical Study on Object Oriented Metrics. Proceedings of the Sixth International Software Metrics Symposium, 242-249.

Tracz, W., (1979). Computer Programming and the Human Thought Process. Software - Practice and Experience, 9, 127-137.

Unger, B., Prechelt, L., (1998). The Impact of Inheritance Depth on Maintenance Tasks - Detailed Description and Evaluation of Two Experiment Replications. Fakultat fur Informatik - Universitaet Karlsruhe, 19/1998.

Vallar, G., Baddeley, A., (1982). Short-Term Forgetting and the Articulatory Loop. Quarterly Journal of Experimental Psychology, 34, 53-60.

van Dijk, T., Kintsch, W. (1983). Strategies of Discourse Comprehension. Academic Press.

von Mayrhauser, A., Vans, A., (1995a). Program Understanding: Models and Experiments. Advances in Computers, 40, 1-38.

von Mayrhauser, A., Vans, A., (1995b). Industrial Experience with An Integrated Code Comprehension Model. Software Engineering Journal, 171-182.

von Mayrhauser, A., Vans, A., (1996). Identification of Dynamic Comprehension Processes During Large Scale Maintenance. IEEE Transactions on Software Engineering, 22:6, 424-437.

von Mayrhauser, A., Vans, A. (1997). Program Understanding Behavior During Debugging of Large Scale Software. Workshop on Empirical Studies of Programmers, 157-179.

von Mayrhauser, A., Vans, M., Howe, D., (1997). Program Understanding Behaviour during Enhancement of Large-Scale Software. Journal of Software Maintenance: Research and Practice, 9, 299-327.

Votta, L., (1993). Does Every Inspection Need a Meeting ? ACM Software Engineering Notes, 18:5, 107-114.

Weiss, S., Kulikowski, C. (1991). Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems. Morgan Kaufmann Publishers.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., Corritore, C., (1999). A Comparison of the Comprehension of Object-Oriented and Procedural Programs by Novice Programmers. Interacting with Computers, 11:3, 255-282.

Wilde, N., Matthews, P., Huitt, R., (1993). Maintaining Object-Oriented Software. IEEE Software, 75-80.

Youden, W., (1950). Index for Rating Diagnostic Tests. Cancer, 3, 32-35.

Zhang, G., Simon, H., (1985). STM Capacity for Chinese Words and Idioms: Chunking and Acoustical Loop Hypotheses. Memory and Cognition, 13:3, 193-201.

Zweig, M., Campbell, G., (1993). Receiver-Operating Characteristic (ROC) Plots: A Fundamental Evaluation Tool in Clinical Medicine. Clinical Chemistry, 39:4, 561-577.